Dines Bjørner
Manfred Broy
Alexandre V. Zamulin (Eds.)

LNCS 2244

Perspectives of System Informatics

4th International
Andrei Ershov Memorial Conference, PSI 2001
Akademgorodok, Novosibirsk, Russia, July 2001
Revised Papers

Springer

Lecture Notes in Computer Science 2244
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Dines Bjørner   Manfred Broy
Alexandre V. Zamulin (Eds.)

Perspectives of System Informatics

4th International
Andrei Ershov Memorial Conference, PSI 2001
Akademgorodok, Novosibirsk, Russia, July 2-6, 2001
Revised Papers

Springer

Volume Editors

Dines Bjørner
Technical University of Denmark, Informatics and Mathematical Modelling
Bldg. 322, 106-108, Richard Petersens Plads, 2800 Lyngby, Denmark
E-mail: db@it.dtu.dk

Manfred Broy
Technical University of Munich, Computer Science Department
Arcisstr. 21, 80290 Munich, Germany
E-mail: broy@informatik.tu-muenchen.de

Alexandre V. Zamulin
A.P. Ershov Institute of Informatics Systems
Acad. Lavrentjev ave., 6, Novosibirsk, 630090, Russia
E-mail: zam@iis.nsk.su

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Perspectives of system informatics : revised papers / 4th International
Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk,
Russia, July 2 - 6, 2001. Dines Bjørner ... (ed.). - Berlin ; Heidelberg ;
New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo :
Springer, 2001
(Lecture notes in computer science ; Vol. 2244)
ISBN 3-540-43075-X

CR Subject Classification (1998): F.3, D.3, D.1, D.2, F.1

ISSN 0302-9743
ISBN 3-540-43075-X Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10845892 06/3142 5 4 3 2 1 0

Preface

This volume comprises the papers presented at the Fourth International Andrei Ershov Memorial Conference "Perspectives of System Informatics", Akademgorodok (Novosibirsk, Russia), July 2–6, 2001. The main goal of the conference was to give an overview of research directions decisive for the growth of major areas of research activities in system informatics.

The conference was held to honor the 70th anniversary of the late Academician Andrei Ershov (1931–1988) and his outstanding contributions towards advancing informatics. It was the fourth conference in the line. The First International Conference "Perspectives of System Informatics" was held in Novosibirsk, Akademgorodok, May 27–30, 1991, the second one on June 25–28, 1996, the third one on July 6–9, 1999. The three conferences gathered a wide spectrum of specialists and were undoubtedly very successful.

The fourth conference included many subjects of the previous ones, such as theoretical computer science, programming methodology, and new information technologies, which are the most important components of system informatics. The style of the three previous conferences was preserved to a certain extent: a number of invited papers were presented in addition to contributed regular and short papers.

This time 73 papers were submitted to the conference by researchers from 19 countries. Each paper was reviewed by three experts, at least two of them from the same discipline as the authors or a closely related one. The reviewers generally provided high quality assessment of the papers and often gave extensive comments to the authors for the possible improvement of the presentation. As a result, the program committee selected 26 high quality papers as regular talks and 22 papers as short talks. Some hot topics in System Informatics were covered by four invited talks given by prominent computer scientists from different countries.

To celebrate the 70th anniversary of the late Academician A. P. Ershov, a special memorial session was organized. It included two invited talks and a number of short informal communications. The invited talks were given by two prominent Russian computer scientists who worked either side by side with A. P. Ershov or in a closely related area.

Andrei P. Ershov was a man for all seasons. He commanded universal respect and received affection all over the world. His view of programming was both a human one and a scientific one. He created at Akademogorodok a unique group of scientists – some now in far away regions of the world: a good example of "technology transfer", although perhaps not one that too many people in Russia are happy about.

Many of his disciples and colleagues continue to work in the directions initiated or stimulated by him, at the A. P. Ershov Institute of Informatics Systems, which is the main organizer of the conference.

We are glad to express our gratitude to all the persons and organizations who contributed to the conference – to the sponsors for their moral, financial, and organizational support, and to the members of local Organizing Committee for their efforts towards making a success of this event. We are especially grateful to N. Cheremnykh for her selfless labor when preparing the conference.

July, 2001

D. Bjørner
M. Broy
A. Zamulin

Organization

Conference Chair

Alexander Marchuk (Novosibirsk, Russia)

Program Committee Co-chairs

Dines Bjørner (Lyngby, Denmark)
Manfred Broy (Munich, Germany)
Alexandre Zamulin (Novosibirsk, Russia)

Program Committee

Janis Barzdins (Latvia)
Frédéric Benhamou (France)
Mikhail Bulyonkov (Russia)
Gabriel Ciobanu (Romania)
Piotr Dembinski (Poland)
Alexander Dikovsky (France)
Uwe Glässer (Germany)
Victor Ivannikov (Russia)
Philippe Jorrand (France)
Leonid Kalinichenko (Russia)
Alexander Kleschev (Russia)
Gregory Kucherov (France)
Sergei Kuznetsov (Russia)
Alexander Letichevski (Ukraine)
Giorgio Levi (Italy)
Dominique Mery (France)
Bernhard Möller (Germany)
Hanspeter Mössenböck (Austria)
Ron Morrison (Scotland)
Valery Nepomniaschy (Russia)
Peter Pepper (Germany)
Francesco Parisi-Presicce (Italy)
Jaan Penjam (Estonia)
Alexander Petrenko (Russia)
Jaroslav Pokorny (Czech Republic)
Wolfgang Reisig (Germany)
Viktor Sabelfeld (Germany)
Don Sannella (Scotland)
Vladimir Sazonov (UK)
David Schmidt (USA)
Igor Shvetsov†[1] (Russia)
Sibylle Schupp (USA)
Nicolas Spyratos (France)
Lothar Thiele (Switzerland)
Alexander Tomilin (Russia)
Enn Tyugu (Sweden)
Andrei Voronkov (UK)
Tatyana Yakhno (Turkey)

[1] Igor Shvetsov was a member of staff at the A.P. Ershov Institute of Informatics Systems, and an internationally know specialist in the field of constraint programming. He prematurely passed away in April 2001. Right up until the last days of his life, he took part in the work of the PSI 2001 Program Committee.

Additional Referees

S. Ambroszkiewicz
M. A. Bednarczyk
A. Bossi
O. Bournez
M. Breitling
D. Cansell
S. Chernonozhkin
D. Colnet
M. Dekhtyar
H. Ehler
Th. Ehm
G. Ferrari
J. Fleuriot
J. Freiheit
K. Freivalds
W. Grieskamp
G. Hamilton
P. Hofstedt
P. Hubwieser
P. Jackson
P. Janowski
N. Jones
G. Klein
K. Korovin
A. Kossatchev
P. Lescanne
F. Levi
I. Lomazova
P. Machado
A. Masini
E. Monfroy
D. von Oheimb
P. Poizat
M. Proietti
A. Rabinovich
A. Riazanov
J. Ruf
Ch. Salzmann
F. Saubion
W. Schulte
B. Schaetz
N. Shilov
F. Spoto
N. Tillmann
J. Tucker
M. Turuani
M. Valiev
W. Vogler
J. Winkowski
J. Vain
M. Veanes

Conference Secretary

Natalia Cheremnykh (Novosibirsk, Russia)

Local Organizing Committee

Sergei Kuznetsov
Gennady Alexeev
Alexander Bystrov
Tatyana Churina
Vladimir Detushev
Olga Drobyshevich
Vera Ivanova
Sergei Myl'nikov
Elena Okunishnikova
Vladimir Sergeev
Anna Shelukhina
Irina Zanina

Sponsors

Support from the following institutions is gratefully acknowledged:

- Russian Foundation for Basic Research
- Office of Naval Research, USA
- Microsoft Research, UK
- xTech, Russia

Table of Contents

A.P. Ershov — A Pioneer and a Leader of National Programming

Igor V. Pottosin

A. P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Acad. Lavrentjev ave., 630090, Novosibirsk, Russia
ivp@iis.nsk.su

April 19, 2001, seventy years had passed since the birth of Andrei Petrovich Ershov. He was a pioneer of programming in this country, and one of its leaders; a scientist with considerable and decisive influence on the development of national programming. I have already tried to overview A. P. Ershov's scientific findings in a paper recently published in a collection of his selected works; this memorial report is primarily intended to tell the story of A. P. Ershov as a pioneer of programming and as its leader for many years.

A. P. Ershov was becoming a scientist just at the time when programming was becoming a science. This is unique to his scientific career, so different from that of his contemporaries who had previously worked in other domains. Not only did he have to be a researcher but also a propagandist, a defender, and an organizer, as this newly created discipline required him to be. His scientific findings and his organizational activities are important both in themselves and for their role in the auto-identification of the new scientific trend and in setting up the framework for its internal research.

1950 was the starting point for programming in this country, when a model of MESM was created, the first computer in the USSR and in continental Europe. Ershov had devoted his life to programming two years later, choosing his major in computational mathematics at the Faculty of Mathematics and Mechanics at Moscow State University. He was in the group of graduates who were the first to be certified as programming specialists in the USSR. Among his fellow graduates were E. Z. L'ubimsky, V. S. Shtarkman, I. B. Zadyhaylo, V. V. Lucikovich, O. S. Kulagina, N. N. Rikko, and others.

It is interesting to look at the origins of the first generation of programmers. In no case they dreamed of devoting their lives to programming from the school bench. As a rule, they came from related branches of science: mathematics, mechanics, physics, engineering; and some teachers of mathematics were also among them. Ershov and his colleagues were distinguished from others as an elite, since only they had a basic education in this new field. Actually, Ershov had dreamed of becoming a physicist and only circumstances urged him to enter the Faculty of Mathematics and Mechanics. Ye. A. Zhogolev, who has contributed to the orientation of Ershov towards programming, remembers that it was Ershov's interest in the physical principles of computers that has led him to study computational mathematics — the only course where these principles were being studied.

Initially this new discipline could not be regarded as a discipline in itself and had to be called by another name. Note: not under an alien name, but under another name. All those who moved to programming were given the label "mathematicians" independently of their original specialty. It is worth noting that the "computing world" of that time roughly consisted of two camps. The people in one camp were labeled "engineers"; those in the other camp were called "mathematicians" and they were in fact, programmers, the people who produced software.

To be incorporated into mathematics was, on the one hand, rather natural. It was necessary to develop under some "umbrella" or other, and the umbrella of mathematics was the most appropriate one: the strictness and accuracy of decisions and the purely intellectual nature of the resulting product characterize both mathematics and programming. But on the other hand, several facets of programming (the importance of pragmatics, impossibility — to this point of time — to deduce everything from proofs alone) distinguish it from mathematics and made it — for some mathematical Puritans — a "dirty" branch of mathematics. I remember hearing M. I. Kargapolov say (rather typically): "We always had mathematics with theorems, and now we have a new mathematics — without theorems." Many mathematicians, especially those working in the field of computational mathematics, thought that the unique role of programmers was to serve demands imposed by computational tasks and that there were not and could not be any internal problems relating to the programming itself.

There was only one way to react to such views: to build a foundation for the new scientific discipline, to develop its models and methods which could testify to the existence of essential problems inherent in programming and thus to promote its auto-identification as a distinct scientific domain.

Being one of the pioneers of programming, Ershov was well acquainted with the problems involved in creating a new discipline. He had finished his candidate thesis dedicated to one model of programs (operator algorithms) in 1959; it was not until 1962 that he could support it. "Pure" mathematicians could not understand its importance: namely, that the model presented adequately reflected essential properties of real programs. Ershov was also permanently opposed while working on his famous Alpha-project. As many people claimed, a team of experienced programmers would be better off writing useful applied programs than a useless compiler which in no case would achieve its declared goal — translating programs written in an Algol-like language into programs whose quality would be comparable with hand-written programs.

There were two possible reactions to these external difficulties in the life of programming. One could "give up": either investigate the purely mathematical problems of programming (they are numerous, thank goodness) or perform computational tasks without even trying to see programming as a domain in its own right. Conversely, one could break ties with mathematics and develop programming systems without any attempt to employ precise or formal methods. The wisdom of Ershov prevented him from taking either of these paths. He simultaneously and in an interconnected way developed both the conceptual foundation of programming and precise mathematical models that formalize as far as possible corresponding concepts. Ershov's school of programming in Novosibirsk is based upon this very combination.

Ershov's first scientific results (and this was also the case for most programming pioneers) were connected with computational tasks. His first paper was the article "One method of the inversion of matrices" ("Doklady Akademii Nauk SSSR", 1955), but Ershov would not have become a programmer, had he not written a standard program for the BESM computer implementing this new method. In fact, the choice of this topic was motivated by Ershov's primary mathematical interest: prior to choosing computational mathematics as his major he intended to specialize in algebra. But being challenged by the internal problems of programming, Ershov, like many other pioneers, moved to completely new, unexplored areas: programming languages and systems were the first among them. Ershov was not the first in that field, but he was one of the early groundbreakers. He was one of the developers of the Programming Program for the BESM computer which was one of the first domestic compilers. His brilliant ideas mainly contributed to the basis for language concepts and compilation methods. Ershov was the first to suggest (at least in this country) such a language construct as a loop and such a method as a hash function. In 1958 he published a well-known book on compiling, the first in the world, later translated into English (1959) and also into Chinese.

Beginning with his very first works, Ershov was recognized as a leader in the field of programming languages and systems. The ideas suggested by Ershov and the experience gained when carrying out his projects — the optimizing compiler Alpha, the cross-compiler Algibr, the Alpha-6 compiler for the Besm-6 computer, and the multi-language compiling system BETA — have contributed to the modern fundamentals of compilation. Memory optimization techniques, the notion of internal language as a semantic representation of programs for their optimization or cross-compilation and unified compilation scheme should be mentioned as Ershov's personal contribution.

Ershov has also greatly contributed to the development of programming languages that succeeded Algol-60: the Alpha-language, an extension of Algol-60, already contained some features appropriate to modern languages, such as multidimensional values, a variety of loop constructs, initial values and so on. His Sigma language is an example of a language kernel extensible with the mechanism of substitution.

Ershov's results on specializing programs by means of mixed computation are widely recognized. Ershov had predecessors in this field (Futamura, Turchin) but it was he who cristallized the principle of mixed computation which later gave an impetus for many theoretical and practical works on mixed computation and partial evaluation. His disciples have produced works on practical and highly efficient mixed computation. His ideas incited some other approaches to specializing programs, such as concretization. Ershov has also formulated the notion of a transformational machine as an approach to constructing reliable and high-quality software. These ideas form the basis of a new trend of programming, namely transformational programming, which looks promising, though as yet has not been practically applied on a wider scale.

A special place in Ershov's works is occupied by the AIST project. I think that the significance and value of this highly innovatory project have been underestimated. In this project Ershov headed the construction of the entire computing system, both its architecture and software. The initial stage of this project, AIST-0, has been one of the first national multi-processor systems with a rich software providing different access modes from batch to time-sharing.

The AIST software was being developed practically in parallel with other projects, both in the USSR and abroad, on constructing powerful operating systems, prototypes of modern OS (Multics, OS IPM and others), and it was remarkable and original enough in comparison with these systems. As to the AIST architecture, it was the first complex architectural project in Novosibirsk, followed later by another original projects carried out in the Institute of Mathematics and in the Computing Center of the Siberian Branch of the USSR Academy of Sciences. The software model had three levels: the kernel of the OS, specialized interactive systems and application software. The project was interrupted and next stages were not realized because of approval of a possibly erroneous national program, about which Edsger Dijkstra once said: "The greatest success of the USA during the cold war was the fact that the USSR accepted the IBM architecture."

The first generation of programmers had the honour of building a conceptual framework, a fundamental basis of the new scientific discipline. They were like newcomers "with the world so new-and-all" (R. Kipling) and they came up with ideas and developed methods by evaluating their own experience. Ershov with his mathematical training and his experience in heading several essentially new projects (some of which have been just mentioned) has done much towards laying foundations of the new discipline. I want to mention one important detail: he paid due attention to the work on terminology. He himself introduced several fundamental terms of the Russian professional language, such as "informatics", "programmnoe obespechenie" (literally "programming provision") for "software", "technology of programming" for "software engineering", etc. It was not only the case of inventing new names. Substantial conceptual and methodological work backed these names.

Let us return to the term "informatics". As said above, Ershov, like all of us, considered himself for a long time as a person working in a peculiar, but still mathematical, field and did not fully realize the uniqueness of his job. It was not easy to leave the habitual umbrella but when specific scientific luggage had been accumulated, recognition of this specificity became inevitable. As soon as Ershov realized that a new discipline had come in existence, he made a decisive step, proposed the new term "informatics", and published the paper "On the Subject of Informatics" where he outlined the subject and the meaning of the new science. Ershov gave to the term "informatics" a wider interpretation than that of its traditional English equivalent "Computer Science". Namely, he understood it as a fundamental natural science studying processes of information transfer and processing. As to "computer science" itself, it is referred to as the "working" up-to-date content of informatics.

No less important was Ershov's coining of the phrase "technology of programming" ("software engineering"). The first generation of programmers considered the process of software construction as a highly intellectual work akin to scientific research. And they were right, to some degree and in the initial stages. Ershov was the first (at least in this country) who saw the other side of programming, the industrial rather than the scientific, as the basis of the new industrial branch, namely the software industry. This point of view was met with sharp objections by many people, and Ershov had to defend his position for a long time and with much tenacity. This position, which later proved to be correct, is reflected by the term "technology" as applied to programming.

Perceiving his leadership, Ershov understood all the responsibility for building the foundation of the new discipline and distinctly saw the needs of the whole field. With respect to this he emphasized two problems. The first one is the problem of creating the "lexicon of programming", that is a conceptual basis, a system of interconnected and formalized notions reflecting at a good level of abstraction everything needed for the process of program construction. A large part of Ershov's work is in this subject. It is enough to recall his works related to the BETA system. The second problem is close to the first; it concerns the development of the theory, of the formal methods, of everything that can now be distinctly traced in the new methodologies and technologies of program design. In this direction Ershov has also done a lot. His contribution to the theory of program schemata will be discussed later, and meanwhile I would like to stress that he ever saw the link between the theory and the practice of programming and knew both how to draw a basis for theoretical models from practical results, and how to apply theoretical research to practical tasks. As early as the Alpha compiler, he was using the theory of memory economy which he and S.S.Lavrov had elaborated to construct corresponding optimizing transformations. This work had further consequences. A large proportion of subsequent works from the Novosibirsk school on creating program models (linear schemata, multiple schemata, linear programs, large-block schemata, regular schemata) and on their application for proving correctness of optimizing transformations sprung from this source. Another example: heading the design of multi-processing and multi-programming system AIST, Ershov simultaneously initiated research on formal models of parallel programs.

In 1977 Ershov published a splendid and, in some respects, unique book "Origins of Programming: Discourses on Methodology". This book showed (this constitutes its special importance) how the investigation of practical problems leads to theoretical models and how the models assist in solving practical problems. Monographs on programming, as we well know, quickly become outdated. But this book appears to be an exception from this sadly common rule: in 1990 it was published in English.

One of the main problems of any new trend is the training of professionals and researchers. With this respect as well, Ershov was a pioneer and leader.

He devoted a lot of time to educational informatics and its methodology, to the writing of textbooks and learning systems, to advertising the importance of these kinds of activity. He was the recognized leader in this area in this country.

On the one hand, he was occupied by the problem of educating professional programmers and he dedicated a series of his works to this. On the other hand, he started the school informatics in this country, and here his activities were multifaceted: lectures on TV, one of the first textbooks on informatics for secondary schools (this textbook has been translated into many languages of the USSR), organization of summer schools of young programmers, organization of a voyage of young programmers to the United States, and others. It is essential that he saw the unity of both sides: of the education of high-level specialists and of upgrading informational culture of the society starting in the school.

Ershov was aware of his leadership and he saw the destiny and development of the new trend and the destiny of people who would work in that field as his own responsibility. He could defend the value of programming and programmers to the outside world. He had an excellent (and not common) feature: he valued the authority of knowledge, of mental outlook, of spirit, but not that of post and position, and he applied this attitude to himself. His statements were always thoughtfully and logically structured, he watched himself attentively and he never imagined that these statements should be taken as indisputable truths simply because they came from an “important person”.

I cannot remember any case in which he, though being aware of his huge influence, took the stand of an oracle, which would have been respected only because he occupied a high position.

I have already mentioned in what degree Ershov’s scientific results contributed to the formation and auto-identification of the new discipline. The definition itself of the spirit of that discipline, of its ethic, of its professional specificity are due to the works and to the activity of Ershov. The social outlook of programming in this country was formed thanks to the activity of such organizations as the Commission on System Software, Council of Cybernetics, Committee on Program Languages and Program Systems. All these bodies were headed by Ershov. His well-known works “Two faces of programming”, “Aesthetics and the human factors of programming”, “Programming, the second literacy” have clearly and vividly described the spirit and uniqueness of this new human endeavor.

Ershov was also a leader in the social life of programmers. His leadership, unconditional and recognized by everyone, is also associated with his personal qualities.

Possessing a true strategy of thought, he predicted the future of the newborn phenomena, clearly seeing “points of growth”. It is sufficient to recollect the works he once initiated: the implementation of one of the first specification languages SETL, technology for creating intellectual systems, theory of parallel programming, construction of the Shkolnitsa (Schoolgirl) system which was a methodologically grounded tool for teaching programming at school. Just from mere appearance of personal computers he understood their great future and figuratively called the first PCs "the ancestors of mammals in the dinosaurs' world of contemporary computers".

I hardly know anyone who could, like Ershov, enjoy the ideas of others. He listened to them attentively, independent of whether they came from a post-graduate student or from a professor, actively propagated them, assisted authors in realizing their ideas. Numerous are those for whom the assistance of Ershov was quite essential. Among them are E. Tyugu (Estonia), M. Tsuladze (Georgia), I. Velbitsky (Ukraine), D. Todoroy (Moldova), A. Terekhov (St.-Petersburg) and many others. To say nothing of us, Siberians.

Thus, Ershov was both one of pioneers and a leader of national programming. While the times he lived in have primarily contributed to the first role, his outstanding personality has mainly contributed to the second one.

A.A. Lyapunov and A.P. Ershov in the Theory of Program Schemes and the Development of Its Logic Concepts

Rimma I. Podlovchenko

Research Computing Center, Moscow State University,
Vorobyovy Gory, Moscow, RU-119899, Russia
rip@vvv.srcc.msu.ru

Abstract. The aim of this paper is to survey the advent and maturation of the theory of program schemes, emphasize the fundamental contributions of A. A. Lyapunov and A. P. Ershov in this branch of computer science, and discuss the main trends in the theory of program schemes.

This paper was written in commemoration of Andrei Petrovich Ershov, who exerted great influence on the development of theoretical programming, the theory of program schemes specifically. The choice of this section for discussion does not only display the author's predilection. The key point is that the concepts introduced by A.P. Ershov as the basis for the theory of program schemes in the time of its coming into being have been consolidated in subsequent years along the trend predicted by him. It was intended that this paper would elucidate the facts.

1 The Formation of Scientific Preferences of Andrei Ershov

This formation took place in the 50s of the past century. The years were commemorated by the appearance and development of domestic electronic computers. There was a need in specialists for their designing and servicing.

Moscow State University responded to the call at once. In 1950 the department of computational mathematics was set up in the mechanico-mathematical faculty. Teaching of the students in numerical analysis was one of the objectives set for the department. Another aim was training the students for using the newly born computers. In contrast to the first one, this task did not have any clear-cut outlines of solution. Initially, it was assumed that the use of computers for solving mathematical problems would necessitate detailed knowledge of the machine design. This consideration influenced much the choice of disciplines included in the curriculum for those studying in the department, namely: radio engineering and electronics, electrical engineering, theory of mechanisms and machines, adding machines and instruments, drawing. The subjects enumerated replaced fully such disciplines as the set theory, higher sections of algebra, functional analysis, mathematical logic (the theory of algorithms was not taught yet in those years).

Naturally, in due course the things that were actually indispensable for the graduates of the department were determined and the curriculum got rid of unnecessary subjects, whereas the mathematical foundation was restored.

Andrei Ershov became a student of the department of computational mathematics in 1952. He was lucky, as the gaps in his mathematical education in the years of his studentship were eliminated with assistance of Alexey Andreevich Lyapunov, who assumed supervision over Andrei Ershov post graduate education. Alexey Andreevich drawn up a program of qualifying examination for the candidate's degree satiated in mathematics and strictly followed its implementation advising personally on the subjects included in the program.

Let us go a couple of years back to 1952, when Alexey Andreevich accepted a position of professor at the department of computational mathematics. The event is noteworthy, as scientific life at the department livened up a lot. Andrei Ershov was then the fourth year student.

His enthusiasm and gift of convincing helped him to turn many of students in the department into his belief in extraordinary future that lied ahead for the machines and programming. In 1952/1953 academic year Alexey Andreevich read the famous course of lectures "Principles of Programming" (the relevant materials in a somewhat revised form were published only in 1958 [12]) to the students in the department. It was the first in the country course in programming that played the fundamental role in the development of a new branch of knowledge, i.e. programming.

A new view on formalization of the concept of algorithm as such was presented, proceeding from convenience of its use when solving practical problems. Meanwhile, the already existent formalizations of the algorithm concept (such as Turing's machines, Markov's normal algorithms) were aimed exceptionally at studying the nature of computations rather than practical application. In the course read by Alexey Andreevich a programming language was suggested, which was precursor of the currently used high-level languages; the language was called the *operator language.* Its introduction made it possible to describe techniques of programming. The operator language and the relevant programming techniques were integrated under the name of the *operator method.*

The operator language was not formalized. However, the problems of programming could be actually discussed, i.e. for the first time programming was treated as a branch of science with its own problems. Two problems were selected by Alexey Andreevich for the principal ones, namely:

- automation of making up programs;
- optimization of programs that were initially made up.

The problems were considered mutually interrelated, though each of them could be studied separately.

Alexey Andreevich attracted students in the department, Andrei Ershov among them, for coping with the first task. He offered that the operator language is used as support one. Construction of the so-called programming program was planned; the program receiving for an input a description of algorithm in the operator language was aimed to transform it into the program executing the algorithm. Conceptually, the programming program was to be assembled from blocks performing individual functions. Andrei Ershov was entrusted with construction of arithmetic block.

The work was the initial step in the studies relating to construction of translators, the studies that ran all through Andrei Petrovich subsequent activities in programming. The works in this trend are enumerated in [7]. The evolution of his ideas and techniques for constructing the translators is discussed in the preface article by I.V. Pottosin [7] . It is worth noting that already in that initial work the idea arose that the memory space of the programs should be optimized (refer to [1]). It was the first manifestation of the global intention — to apply optimization techniques to the made up programs along the process of translation.

2 Early Investigations in Theoretical Programming

They were directed by A.A. Lyapunov and pertained to the second problem. A.A. Lyapunov started from the following requirement: mathematical solution of the program optimization problem should actually rely on strict definition of the program as such, its structure and functioning, specifically. Only then one can speak of the function computed by the program and, accordingly, introduce the concept of *functional equivalence* of programs by using the requirement of coincidence of the functions realized by the programs. Optimization of a program is performed by means of its *equivalent transformations* (e.t.), i.e. transformations, which retain the function realized by the program. Thus, the task of constructing the program e.t. is brought to the forefront after the program formalization.

Alexey Andreevich assumed that program formalization may be performed on the base of the operator language. It is in this way that the task was formulated for Andrei Ershov. Then A. Ershov was his post graduate.

Intuition prompted that the program formalization taking a full enough account of actual program properties, is a new universal definition of an algorithm. This fact was established by A.Ershov in 1958 and published in [2,3]. He introduced the concept of operator algorithm and proved that all Markov's normal algorithms are realized by the operator algorithms.

We shall turn now to the task of constructing the program e.t. According to mathematical tradition this task is treated as follows: one shall find an e.t. system, which is complete in a given class of programs. The *completeness of an e.t. system* implies that for any two equivalent programs belonging to the given class there exists a finite chain of transformations belonging to the system that transforms one program into the other. Clearly, such system is the best solution of the task considered. As only solvable e.t. systems are of practical interest, their search ranks as *e.t. problem.* But then the necessary condition for its positive solution is decidability of the *equivalence problem*; this means that there should exists an algorithm that recognizes the equivalence of programs.

Note that approximate solutions of e.t. problem, i.e. designing of incomplete e.t. systems that could be used in practice, are of much interest as well.

A.A. Lyapunov assumed that one of possible ways of its solution is in constructing e.t. using not the programs as such, but their models, i.e. program schemes. The logic schemes of the programs he considered in two ways: as an algorithm description and as an algorithm scheme description. In the first case all operators used in the logic scheme and logic conditions are made specific. In the second case their specific definition is omitted, only the places occupied by them being fixed and a relation of scheme equivalence is defined.

In general, A.A. Lyapunov intention was as follows. As the structures of program and its scheme coincide, each transformation of the scheme is the transformation of the program, modeled by the scheme. Let us assume that the equivalence of schemes implies the equivalence of programs modeled by the schemes (The first equivalence is called the equivalence *approximating* the second one). In this case, if scheme transformation is equivalent, then it is equivalent for the program modeled by the scheme, as well. Thus, each e.t. system for schemes is the same for the programs, being the approximate solution of program e.t. problem. And the best solution is an e.t. system, which is complete in the set of program schemes. So, the scheme e.t. problem is converted to the fundamental problem for program schemes. And here, as for programs, the necessary condition of its positive solution is the decidability of scheme equivalence problem in given class of schemes.

The following task was formulated by A.A. Lyapunov for his post graduate Yu.I. Yanov: to define approximate equivalences in the set of operator schemes, introduced by A.A. Lyapunov, and to solve for them the scheme e.t. problem, considering some class of schemes. The investigation, executed by Yu.I. Yanov in [27] offers the new scientific direction, which was named theory of *program schemes*, and the operator schemes turned to *Yanov schemes*.

Let us consider in more details this first result obtained in scheme theory by taking the formal concept of Yanov schemes improved by A.P.Ershov in [4] and labeling functions as they were modified by the author in [16].

Program schemes are defined over some finite alphabets Y and P. The elements from Y and P are called *operator symbols* and *logical variables* respectively (each variable can be evaluated either by 0 or by 1).

A *program scheme* (or simply scheme) is represented by a finite directed graph. Two nodes of the graph are distinguished: the *entry node*, which has only one outgoing edge and has no incoming edges, and the *exit node*, which has no outgoing edges. The other nodes of the graph are either *transformers* or *recognizers*. Each transformer has one outgoing edge and is associated with an operator symbol from Y. Each recognizer has two outgoing edges marked with 0 and 1; it is associated with a logical variable from P.

An example of a program scheme is depicted in Fig. 1. This scheme corresponds to the algorithm that computes $n!$ for $n > 0$ (see Fig. 2).

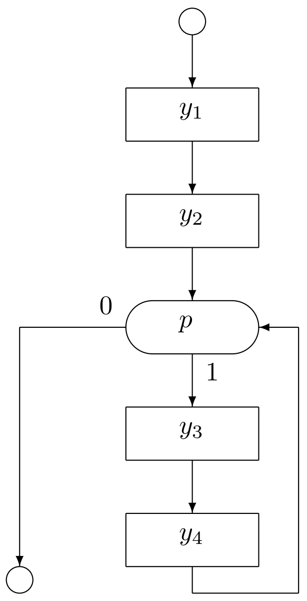


Fig. 1

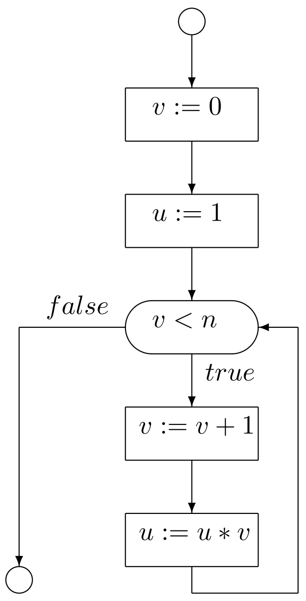


Fig. 2

The functional description of a scheme is related to the process of its execution, which consists in traveling through the scheme accompanied by the accumulation of a chain of operator symbols. The corresponding path is determined by a priori given *labeling function*, which is defined as follows.

Let

$$X = \{x \mid x : \ P \ \to \ \{0, 1\}\}.$$

The elements of X are sets of values of all logical variables. Words in the alphabet Y will be referred to as *operator chains*. The labeling function is a mapping of the set Y^* consisting of all operator chains into the set X. Denote by $\mathcal{L}$ the set of all labeling functions.

Let G be a scheme and μ be a function from $\mathcal{L}$. The execution of the scheme G on the function μ begins at the entry node of the scheme with the empty operator chain and consists in tracing the scheme. The passage through a transformer is accompanied by adding from the right an operator symbol corresponding to this transformer to the current chain. The passage through a recognizer does not change the current chain. Let h be the current chain and p be a variable assigned to the recognizer. Then, the value of the variable p is extracted from the set μh, and the tracing is continued along the edge marked by this value. The scheme execution is completed when the process reaches the exit of the scheme. In this case, the scheme G is said to *stop* on μ, and the *result of its execution* is the operator chain accumulated.

In [27] Yu.I. Yanov considered a parametric set of equivalences of schemes over basis Y, P. Parameter denoted by s was called a *shift distribution* in Y; it induces the set of labeling functions denoted by L_s. By definition schemes G_1 and G_2 are equivalent for the given s, if and only if they stops on the same functions from L_s and chains obtained for a given function coincide. The fundamental result of [27] is Theorem 1.

Theorem 1. *Whatever is a shift distribution in Y for the equivalence induced by them both problems (of equivalence and of e.t.) are solved in the class of schemes over Y, P, where each operator symbol occurs at most once.*

Decidability of both problems for arbitrary schemes over Y, P was proved later in [25].

3 To the Characteristic of First Results in Theoretic Programming

The result obtained by A. Ershov in [2,3], placed the proposed formalization of a program among other formalizations of an algorithm, the mere fact of it being important. Besides, he changed the attitude towards the construction of program e.t. using their schemes. Really, all computable functions are realized by the programs, hence, the equivalence problem is not decidable for them, the fact being mentioned in [24]. This fact implies undecidability of e.t. problem in the set of all programs. Thus, the following alternative arises: either to find classes of programs with decidable equivalence problem and search for them an exact solution of e.t. problem or to restrict oneself to its approximate solution. So, the modeling of programs by schemes takes on the status of practically necessary task.

Let us discuss the results obtained by Yu. I. Yanov in [27].

Intuition prompted that the equivalences proposed above approximate the functional equivalence of programs. To prove this hypothesis one shall define the concept of program as such.

Below we introduce two notions: a semantics of basis Y, P and an abstract program induced by them.

A *semantics of basis* Y, P denoted by σ is a complex consisting of

- a set Σ_σ; the elements of Σ_σ are called *states*;
- functions $\sigma y : \Sigma_\sigma \rightarrow \Sigma_\sigma,\ y \in Y$;
- predicates $\sigma p : \Sigma_\sigma \rightarrow \{0, 1\},\ p \in P$.

The process of executing a scheme G on pair $(\sigma, \xi_0),\ \xi_0 \in \Sigma_\sigma$, is defined as follows. It begins at entry node of the scheme with the state ξ_0 and consists in tracing the scheme. The passage through a transformer with symbol y is accompanied by transformation of the current state ξ into state $\sigma y(\xi)$. The passage through a recognizer with variable p does not change the current state ξ; the tracing is continued along the edge marked by $\sigma p(\xi)$. The scheme execution is completed when the process reaches the exit of the scheme and then the current state is the *result of the execution*. Scheme G accompanied with a semantics σ

is called an *abstract program*; this program computes the function which maps the set Σ_σ into itself. Two abstract programs are *equivalent* if and only if they realize the same function.

Now, for example, let us take the Yanov scheme equivalence induced by the shift distribution s, for which $L_s = \mathcal{L}$, being called a *strong equivalence.* The fact that the strong equivalence approximates the equivalence of abstract programs is established in Theorem 2.

Theorem 2. *Two schemes over basis Y, P are strongly equivalent if and only if they induce equivalent abstract program for any semantics of Y, P.*

For the first time the fact was established in [14].

The investigations carried out by Yu.I.Yanov precede the advent of the finite automata notion. The connection between the Yanov schemes and a finite automata is given in the following.

Theorem 3. *Whatever is a shift distribution in Y, the equivalence induced is reduced to the equivalence of finite automata.*

For a strong equivalence this fact is established in [25]; for the others it follows from [21].

4 On Some Other Tasks of Theory of Program Schemes

We shall mention from the first the task of an economy of program memory, that was considered by A.P. Ershov, and then the tasks to which he called attention.

The variables used by computer program are beyond the abstraction level of Yanov schemes. To deal with the issue of program memory space optimization new class of schemes of special kind was introduced. By using these schemes it is possible to represent explicitly program data-flow, since for each action the variables-arguments and the variables-results are specified. Informal description of such scheme was presented by A.P. Ershov in [1], formal definitions were suggested later by S.S. Lavrov (see [9]). The contribution of A.P.Ershov and S.S.Lavrov to the solution of the problem of program memory space optimization is described in detail in [6].

A.P. Ershov induced also active investigations of another kind of schemes, namely, *standard schemes* of programs [5]. In such schemes variables occurred program are represented explicitly, but unlike Lavrov schemes above the equivalence and e.t. problems for standard schemes are considered as principal ones. The investigations of standard schemes are expounded very fully in [8].

Extra attention to standard schemes is due to their generic property: most positive results and algorithms obtained for standard schemes can be transferred immediately with slight modifications to real computer programs. To make sure of this we shall recall informally the concept of standard scheme.

Programs modeled by standard schemes are ALGOL-like ones, in which the description of variables, input- and output data are skipped. The programs are constructed over the basis consisting of assignment operators and Boolean expressions by using all known operations of operator composition except for the procedure operator. An example of such a program is given in Fig. 2. The transfer from a program to a standard scheme is realized by replacement of concrete operations and relations used in basis operators of assignment and, accordingly, in Boolean expressions by functional and predicate symbols respectively; these symbols retain the arity of operations and relations. The constructions obtained from concrete operators and Boolean expressions are called *operators over memory* and *predicates over memory*. The structure of a scheme coincides with the control-flow of the corresponding program inducing the scheme. Fig. 3 provides the standard scheme constructed for the program depicted in Fig. 2.

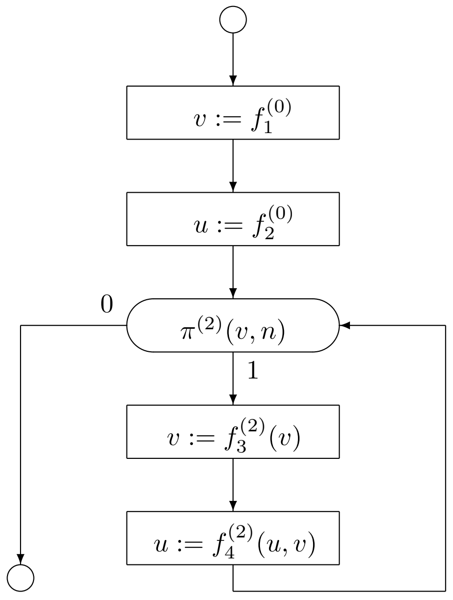


Fig. 3

Now we shall describe how standard schemes operates. Let i be an interpretation of functional and predicative symbols, replacing concrete operations and relations. Interpretation i translates the scheme to the *i-program*. The function realized by the i-program is defined by the same rules that define the function realized by the initial program.

Two standard schemes are *equivalent* if and only if for any interpretation i they give i-programs realizing the same function for the given interpretation. And, as there is an interpretation transforming the scheme into the initial program among interpretations i, the equivalence of standard schemes approximates the functional equivalence of programs modeled by these schemes. This fact ensures that the results obtained for standard schemes according to equivalence and e.t. problems are applicable to real programs.

Describing contemporary state of theory of program schemes in [5] A.P.Ershov posed some problems of the theory. The equivalence problem has an important place among other problems.

The study of standard schemes began with *displaying some certain classes of schemes, for which the equivalence problem is undecidable.* This was established in [10,13]. Therefore it was essential to find out the classes of schemes for which the equivalence problem is decidable. The approaches to the research on this task are described in [5,8]. These approaches are based on imposing some specific requirements on a semigroup of basic operators or the scheme structure. Both approaches seem quite natural. But A.P.Ershov described also some new implicit approach. In essence it is as follows. The functional equivalence considered above is defined in terms of final results computed by the schemes under consideration. A.P Ershov introduced the concepts of history of scheme computation and equivalence relations on the histories. This new type of scheme equivalence is worthy to be a subject of further research if and only if it is stronger than functional equivalence (for example, the logic-term equivalence, discussed in [5], satisfies the requirement).

In [15] it was ascertained that *equivalence stronger than functional is reducible to the functional equivalence in some certain subclass of schemes.*

In fact, for logic-term equivalence this reduction was noticed in [26].

A.P. Ershov attracted attention to the following problem: for program schemes specific approach has to be developed to construct complete e.t. systems. Theory of program schemes starts from mathematical logic, where complete systems are constructed traditionally. The means used by mathematical logic are as follows: a formal calculus is designed; its objects are formulae; in each inference rule only formulae are used. But in case of schemes the role of formula is assumed either by a scheme or a fragment of scheme, i.e. object not being a scheme. Therefore, the task mentioned above arises. An example of new approach was demonstrated by A.P.Ershov in [4]. The author position will be elucidated below.

5 On the Logical Foundations of Program Scheme Theory

They are based on the ideas developed by A.A.Lyapunov, i.e. *the program schemes are created for construction of program e.t.*, and were supported by Yu. I. Yanov and A.P. Ershov in the works, we mentioned above. Being specified as concepts, they are as follows.

Concept 1. Formalization of a program, the description of the program structure and functioning, as well as definition of the program equivalence, is the initial point for constructing the program schemes.

Concept 2. The definition of a program scheme consists in the description of its structure and functioning and the introduction of the scheme equivalence. These components obey the following requirements:

A. The structure of the scheme coincides with the structure of the program modeled by the scheme;

B. The equivalence of the schemes implies the equivalence of programs modeled by the schemes, i.e. equivalence of schemes approximates the equivalence of programs.

These conditions follow from the primary concept. In fact, when the requirement a). is satisfied, each transformation of the scheme is the transformation of the program modeled by the scheme. The requirement b). guarantees that any transformation which preserves the equivalence of schemes is the equivalent transformation of programs.

Concept 3. The e.t. problem for schemes is the principal problem in the theory.

Concept 4. The modeling of program by schemes must be multiciphered, that is reached by means of variety of program scheme equivalence. Really, as the strict solution of scheme e.t. problem is approximate solution of program e.t. problem, then it is necessary to have the spaciousness for the choice of a suitable solution.

Concept 5. The problem of scheme equivalence is the fundamental one. Let us remind that decidability of the equivalence problem in a class of schemes is the necessary condition for searching an e.t. system complete in the class.

6 The Developing of the Logical Concepts of Program Scheme Theory

We shall proceed with the branch of program scheme theory, called *algebraic theory of sequential program models* (or simply it theory of program models). The programs considered in this theory are ALGOL-like ones. They are classified by using/non-using operator procedure (see [19,16] respectively). Below the facts are expounded relating to schemes of programs without procedure operator [16, 17,18].

In this case the formalized programs are the abstract programs, introduced above and constructed over basis Y, P. It is assumed that the semantics of basis Y, P is arbitrary.

The structure of a program scheme coincides with the structure of the Yanov scheme. Hence, condition a). of Concept 2 is met.

According to Concept 4 in the set of program schemes there is a parametric set of equivalences introduced [16]. It expanded essentially the equivalence set considered by Yu.I.Yanov. Each equivalence is induced now by two parameters, namely by

– equivalence ν in Y^*;
– subset L, where $L \subseteq \mathcal{L}$.

(ν, L)*-equivalence* of schemes G_1 and G_2 is defined as follows: for any function from L each time when one of G_1, G_2 stops on this function, the other one also stops, and the results of their execution are two ν-equivalent operator chains.

The set of schemes over Y, P with the given (ν, L)-equivalence is called (ν, L)-*model of programs.*

Following condition b). of Concept 2, it is necessary to select approximate equivalences from the set of all (ν, L)-equivalences. Any of them is called *strict approximating equivalence* if there is a non-empty set S of semantics of basis Y, P, so that for any schemes G_1, G_2 over Y, P the following assumption holds:

> G_1, G_2 are equivalent, if and only if for every semantics σ from S they are equivalent abstract programs on σ.

In [17] the following theorem is proved.

Theorem 4. *Semigroup equivalence is a sufficient condition for (ν, L)-equivalence of scheme over Y, P to be the strict approximating one.*

By definition, (ν, L)-equivalence is a *semigroup equivalence*, if ν and L satisfy the following requirements:

A. For any operator chains h_1, h_2, h_3, h_4 from Y^*

$$(h_1 \overset{\nu}{\sim} h_2) \;\&\; (h_3 \overset{\nu}{\sim} h_4) \;\Rightarrow\; (h_1h_3 \overset{\nu}{\sim} h_2h_4); \tag{1}$$

B. L consists of *ν-coordinated functions*; a labeling function μ is called ν-coordinated if it satisfies the following condition: for any h_1, h_2 from Y^*

$$h_1 \overset{\nu}{\sim} h_2 \;\Rightarrow\; \mu h_1 = \mu h_2;$$

C. L is *closed with respect to shift operation*, i.e. whatever are a function μ from L and chain h from Y^*, the labeling function μ' defined by the identity

$$\mu' g = \mu h g, \;\; g \in Y^*,$$

also belongs to L.

We shall interpret $h_1 \overset{\nu}{\sim} h_2$ as the assertion "h_1, h_2 are ν-equivalent operator chains".

Note that the equivalences of discrete processors discussed by A.A. Letichevsky in [11] are the semigroup equivalences.

As the strict approximation is stronger than the simple approximation defined above, the following question arises: can we possibly lose practically useful equivalences? The answer is given by Theorem 5.

Let us consider the programs given in the formalization used in standard schemes. Denote by K the class of such programs constructed over a finite basis of assignment operators and Boolean expressions. According to definition given above, by transition from programs of class K to standard schemes corresponding to the programs each assignment operator is replaced by operator over memory, each Boolean expression is replaced by predicate over memory. Denote by K_1 the standard scheme class obtained.

Suppose the equivalence in K_1 is induced by such equivalence of programs from K which demands coincidence of program execution results on each variable used by programs. Further, if the assignment operators are replaced by operator symbols and the Boolean expressions are replaced by logical variables, then we obtain the Yanov schemes from the programs of K. Denote this class by K_2. Let us now introduce the correspondence between the operator over memory (the predicates over memory) used in K_1 and the operator symbols (the logical variables) used in K_2 so that their prototypes coincide. In this way the correspondence between the schemes of K_1 and schemes of K_2 is established.

Theorem 5. *It is possible to define a semigroup (ν, L)-equivalence in K_2 so that schemes from K_1 are equivalent if and only if the corresponding schemes from K_2 are (ν, L)-equivalent.*

One of the main advantages of the semigroup equivalences is that they *factor out the equivalences of standard schemes*. But there are some other advantages.

By definition, (ν_1, L_1)-equivalence is *approximated* by (ν_2, L_2)-equivalence, if the latter implies the former. Suppose it takes place. Let us consider a class of abstract programs. Suppose their equivalence is approximated by (ν_i, L_i)-equivalences, $i = 1, 2$, and for both equivalences complete e.t. systems exist, T_1 is the system for the first, T_2 is the system for the second equivalence. Both systems are not complete in the class of programs but T_1 is richer than T_2.

Thus, in the set of program models we may improve the system of e.t. program not leaving this set. The possibility gives rise to the task: to search for sufficient indications for approximating one equivalence by another. The problem mentioned is considered in [23].

Now we turn our attention to the equivalence problem for schemes. The latest survey on this topic was presented by V.A. Zakharov in the conference MCU'2001 and published in [30]. Hence, we restrict our consideration by the novel approaches to this problem and discuss the most significant results obtained so far.

Let us emphasize that here we talk about the theory of program models only. One of the new aspects in studying the equivalence problem is systematic search for algorithms that besides checking the equivalence of a program scheme do it in a reasonable time. The point is that the early investigations were focused mostly on the decidability/undecidability of the equivalence problem and computational complexity of decision procedures was ignored very often. Decision procedures, whose time complexity is exponential, of the size of schemes under consideration were regarded as workable though quite inapplicable in practice. Since only those algorithms, whose time complexity is polynomial of the size of inputs are acknowledged to be efficient, the question arises as to whether it is possible to find out such algorithms by revising the known decidable cases and attacking new variants of the equivalence problem.

Nowadays two novel techniques for designing efficient equivalence-checking algorithms are developed. Both methods go back to [20,29].

The essentials of the first method are presented in Theorem 6 below.

Let us introduce some basic concepts.

It is worth noting that the set Y^* of terms along with concatenation operation may be thought of as a finitely generated semigroup. Its elements are generated by basic actions $y,\ y \in Y$; the empty term λ stands for its neutral element.

Let ν be an equivalence relation on Y^*, and L be the set of ν-coordinated functions from L. In this case we will say that (ν, L)-equivalence is the *equivalence w.r.t. the semigroup Y^* supplied with ν.*

A semigroup Y^* supplied with ν is called *length-preserving*, if Y^* meets all requirements (1) (see theorem 4) and, moreover, whenever h and g are ν-equivalent chains, then they have the same length.

Given a length-preserving semigroup Y^* and a chain h in Y^*, we denote by $[h]$ the equivalence class of h w.r.t. ν, and by $|h|$ the length of h. Clearly, the set

$$E = \{\langle [h_1], [h_2] \rangle \ : \ |h_1| = |h_2|,\ h_1, h_2 \in Y^*\}$$

is also a finitely generated semigroup.

We consider some finitely generated semigroup W, which has $\circ$ for binary operation and e for the neutral element. Suppose that U is a semi-subgroup of W, and w^+, w^* are some distinguished elements in W. Then a quadruple

$$K = \langle W, U, w^+, w^* \rangle \tag{2}$$

is called a *criterial system* for a length-preserving semigroup Y^*, if there exists an integer k_0 and a homomorphism φ from E to U, which satisfy the following requirements:

C1.
$$[h_1] = [h_2] \ \Leftrightarrow \ w^+ \circ \varphi(\langle [h_1], [h_2] \rangle) \circ w^* = e$$
holds for all pairs of chains h_1, h_2;

C2. For every w from $U \circ w^*$ there exist at most k_0 left inverse elements from $w^+ \circ U$, i.e. the equation
$$z \circ w = e$$
has at most k_0 solutions z of the form $w^+ \circ u$, where $u \in U$.

Then we arrive at

Theorem 6. *Suppose that a length-preserving semigroup Y^* has a criterial system (2) such that the identity problem "$w_1 = w_2$?" is decidable in time $t(m)$, where $m = \max(|w_1|, |w_2|)$. Then the equivalence problem for schemes w.r.t. this length-preserving semigroup is decidable in time*

$$c_1 n^2 (t(c_2 n^2) + \log n),$$

where n is the size of schemes to be analyzed, whereas c_1, c_2 are constants that depend on k_0, the number of elements in Y and logic variables in P, and on homomorphism φ.

This theorem, as well as its application to some specific length-preserving semigroup is presented in [29]. One of such semigroup operators, namely free commutative one, was studied earlier in [20]. A free commutative semigroup Y^* is characterized by the following property: chains h_1 and h_2 are equivalent if for every y in Y the numbers of occurrences of y in h_1 and h_2 is the same. The equivalence of schemes w.r.t. free commutative semigroup is decidable in time $cn^2 \log n$, where n is defined as in Theorem 6, whereas c depends on the number of logical variables in P only.

The technique used in [29] is based on the study of algebraic properties of semigroup Y^* supplied with the equivalence of chains.

An alternative approach was introduced by the author [22].

It is based on the computation of invariants for equivalent schemes. By applying this method the following result was obtained in [22].

Theorem 7. *The equivalence of schemes w.r.t. semigroup Y^*, which is both left- and right-contracted is decidable in polynomial time.*

By a left-(right-) contracted semigroup we mean any length-preserving semigroup Y^* for which the supplied equivalence ν is decidable and which satisfies the following properties

$$hh_1 \overset{\nu}{\sim} hh_2 \Rightarrow h_1 \overset{\nu}{\sim} h_2$$
$$h_1h \overset{\nu}{\sim} h_2h \Rightarrow h_1 \overset{\nu}{\sim} h_2$$

It should be noted that a free commutative semigroup is both left- and right-contracted.

Now we think that the task of attracting attention to new trends in studies of equivalence problem is completed and we turn to the presentation of the latest achievements in the research on equivalent transformations in the framework of program models [22].

The means applied by us for construction of complete e.t. system involve the development of suitable formal calculus as before. Its formulae are pairs of scheme fragments. The scheme fragment is defined according to [28], any scheme is considered as a fragment of some specific type.

The fragments belonging to the pair must be coordinated. Only then it is possible to replace in a scheme one of such fragments by the other and result of this replacement will be the scheme again.

So each pair of fragments, when coordinated, induces the set of scheme transformations. In case when this set consists of equivalent transformations only, the fragments belonging to the given pair are called equivalent.

A calculus of scheme fragment pairs must satisfy two requirements:

1. it has the unique inference rule which is the replacement of one fragment by the other; each axiom is a decidable set of equivalent fragment pairs;
2. for each equivalent schemes G_1, G_2 there exists a finite sequence of transformations induced by axioms that transforms the pair (G_1, G_2) to the pair (G_2, G_2).

The formal calculus described (and e.t. system induced by it) is called *finite* if only finitely many axioms are used.

It should be noticed that the cases, when an axiom contains all pairs of equivalent schemes and when an axiom inserts an additional labeling to a scheme (as it takes place in [4]) do not fall under this definition. In [22] by using invariants of equivalent schemes we prove the following

Theorem 8. *If the equivalence of schemes over Y, P is induced by a semigroup Y^*, which is both left- and right-contracted, then there exists a finite e.t. system, which is complete in this set of schemes.*

This generalizes many known results on equivalent transformations.

We conclude our paper with the following summary: the development of the program scheme theory realized in program model theory lends credence to the fruitfulness of its basic concepts. Furthermore, program schemes fit naturally into the row of computational models both by main research problem and by inter-reducibility of these problems.

References

1. A.P.Ershov. On programming of arithmetic operations, *Communications of the ACM*, 1958, **1**, N 8, p.3–6.
2. A.P.Ershov. Operator algorithms. 1. Basic conceptions, In *Problemy kibernetiki*, 1960, **3**, p.5-48 (in Russian).
3. A.P.Ershov. Operator algorithms. 2. Description of basic constructions of programming, In *Problemy kibernetiki*, 1962, **8**, p.211–233 (in Russian).
4. A.P.Ershov. Operator algorithms. 3. On operator schemata of Yanov, In *Problemy kibernetiki*, 1968, **20**, p.181–200 (in Russian).
5. A.P.Ershov. Theory of program schemata, In *Proc. of IFIP Congress'71*, Ljubljana, 1971, p.93–124.
6. A.P.Ershov. Introduction to theoretical programming, Moscow, Nauka, 1977 (in Russian).
7. A.P.Ershov. Selected works, Novosibirsk, Nauka, 1994 (in Russian).
8. V.E.Kotov, V.K.Sabelfeld. Theory of program schemes, Moscow, Nauka, 1991 (in Russian).
9. S.S.Lavrov. On the memory optimization for the closed operator schemes, In *Journal of computational mathematics and mathematical physics*, 1961, **1**, N 4, p.687–701 (in Russian).
10. A.A.Letichevsky. Functional equivalence of finite transformers. II, *Cybernetics*, 1970, N 2, p.14–28 (in Russian).
11. A.A.Letichevsky. On the equivalence of automata over semigroup. *Theoretic Cybernetics*, 1970, **6**, p. 3–71 (in Russian).
12. A.A.Lyapunov. On logical program schemata, In *Problemy kibernetiki*, 1958, **1**, p.46–74 (in Russian).
13. M.S.Paterson. Programs schemata, Machine Intelligence, Edinburgh: Univ. Press, 1968, **3**, p.19–31.
14. R.I.Podlovchenko, G.N.Petrosyan, V.E.Khachatryan. The interpretations of algorithm schemes and different types of scheme equivalence, In *Izvestya Armyanskoy Akademii Nauk*, 1972, **7**, N 2, p.140–151 (in Russian).

15. R.I.Podlovchenko. On correctness and essentiality of some Yanov schemas equivalence relations, *Lecture Notes in Computer Science*, 1975, **32**.
16. R.I.Podlovchenko. Hierarchy of program models, *Progammirovanie*, 1981, N 2, p.3–14 (in Russian).
17. R.I.Podlovchenko. Semigroup models of programs, *Programmirovanie*, 1981, N 4, p.3–13 (in Russian).
18. R.I.Polovchenko. The program models over structured basis, *Programmirovanie*, 1982, N 1, p.9–19 (in Russian).
19. R.I.Podlovchenko. Recursive programs and hierarchy of their models, *Programming and Computer Software*, 1991, **17**, N 6, p.336–341.
20. R.I.Podlovchenko, V.A. Zakharov. A Polynomial-time algorithm that recognizes the commutative equivalence of program schemes, *Doklady RAN, Mathematics*, 1998, **58**, N 2, p.306–309 (in Russian).
21. R.I.Podlovchenko. From schemes of Yanov to the theory of program models. In *Mat. voprosy kibernetiki*, 1998, **7**, p.281–302 (in Russian).
22. R.I.Podlovchenko. On one mass decision of the problem of equivalent transformations for the program schemes, *Programming and Computer Software*, 2000, **26**, N 1, p. 44–52; 2000, **26**, N 2, p.53-60.
23. R.I.Podlovchenko, S.V.Popov. The approximating relation on a program model set, *Vestnik Moscovskogo Universiteta, Computational Mathematics and Cybernetics*, 2001, N 2, p.39–49 (in Russian).
24. H.G.Rice. Classes of recursively enumerable sets and their decision problems, *Transactions of American Mathematical Society*, 1953, **89**, p.25–59.
25. J.D.Rutledge. On Ianov's program schemata, *Journal of the ACM*, 1964, **11**, p.1–9.
26. V.K.Sabelfeld. An algorithm for deciding functional equivalence in a new class of program schemata, In *Theoret. Comput. Sci.*, 1990, **71**, p.265–279.
27. Yu.I.Yanov. The logical schemes of algorithms, In *Problems of cybernetics*, Pergamon Press, New-York, 1960, p.82–140.
28. Yu.I.Yanov. On local transformations of algorithm schemes, In *Problemy kibernetiki*, 1968, **20**, p.201–216 (in Russian).
29. V.A.Zakharov. An efficient and unified approach to the decidability of equivalence of propositional program schemes, *Lecture Notes in Computer Science*, 1998, **1443**, p.247–259.
30. V.A. Zakharov. The equivalence problem for computational models: decidable and undecidable cases, *Lecture Notes in Computer Science*, 2001, **2055**, p.133–153.

The Abstract State Machine Paradigm: What Is in and What Is out

Short Abstract

Yuri Gurevich

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
gurevich@microsoft.com

The computing science is about computations. But what is a computation? We try to answer this question *without* fixing a computation model first. This brings up additional foundational questions like what is a level of abstraction? The analysis leads us to the notion of abstract state machine (ASM) and to the ASM thesis:

> Let A be any computer system at a fixed level of abstraction. There is an abstract state machine B that step-for-step simulates A.

In the case of sequential computations, the thesis has been proved from first principles; see ACM Transactions on Computational Logic, vol. 1, no. 1 (July 2000), pages 77–111. Of course ASMs are not necessarily sequential. In a distributed ASM, computing agents are represented in the global state. New agents can be created, and old agents can be deactivated. There could be various relations among agents and various operations on agents. The global state is a mathematical abstraction different from the conventional shared memory; it may be, for example, that the agents communicate only by messages. The moves of different agents form a partially ordered set. Concurrent moves cause consistent changes of the global state.

Often a formal method comes with a reasoning system. If this is your idea of a formal method then the ASM approach is not a formal method. It is system informatics where modeling is carefully separated from formal reasoning. Notice that formal reasoning is possible only when the raw computational reality is given a mathematical form; ASMs do the modeling job.

The separation of modeling and reasoning concerns does not undermine the role of reasoning. The ASM approach is not married to any particular formal reasoning system and is open to all of them. It is usual for ASMs to have integrity constraints on states. ASM programs can be enhanced with various pre and post conditions. ASM-based testing can be enhanced with model checking. The most important direct application of ASMs is their use as executable specifications. This makes (totally as well as partially) automated reasoning relevant.

For more information on abstract state machines see the academic ASM website

`http://www.eecs.umich.edu/gasm/`.

On Algorithmic Unsolvability

Svyatoslav S. Lavrov

Institute of Applied Astronomy
Russian Academy of Sciences
8, Zhdanovskaya Str., St.Petersburg, 197110, Russia
ssl@lvr.usr.pu.ru

Abstract. Many computational problems are algorithmically unsolvable. How well this fundamental assertion of computability theory is based — that is the main question considered in the paper from a programmer's view.

Introduction — Where Does the Border between Natural and Formal Languages Lie?

To begin with — a comment to the Russell's 'village barber' paradox. A man has many aspects. When at home he eats, sleeps, in the morning washes and possibly shaves himself and after breakfast goes to the work. At work he being the barber receives his clients, cuts their hair and shaves them. He would violate his promise only if he sat in the barber's chair and simultaneously stood nearby and shaved the man sitting there.

One may oppose that all said above is a game with notions taken from the human mode of life and reflected in natural language. However G. Cantor himself expressed the concepts of the set and membership using the words like "collection", "intuition", "intellect", "the whole (indivisible)", which differ from the common ones maybe by a slightly higher style. He simply has had no other means just as we have not got them still. The chance for a set to be its own member is by no means better than for a barber to receive himself as a client.

The attempts taken at the first half of the XX century to remove contradictions from the set theory have led to the seemingly successful creation of the axiomatic set theory. The proper classes introduced in the theory serve as substitute for all ugly sets (undesirable barbers). Since then Cantor's set theory got the attribute "naïve". I try to show that the traditional computability theory has the touch of naïvety too.

The main trouble with the axiomatic approach is in lacking of a model for the whole set theory. The theory with proper classes may be considered as a meta-theory for the one of the common sets. It contains the class playing the role of a subject domain for the latter theory. What may serve however as such a domain for the meta-theory? Just in this vicinity the fuzzy border lies between natural language together with common sense and formal means for expressing the scientific concepts.

In my opinion the problem to be discussed lay on the informal side of the border.The famous Church's thesis says that all known ways to formalize the algorithm notion are essentially equivalent with each other and that they all correspond to the intuitive notion of effectively computable function. Mind that in the second part of this statement the correspondence is considered as merit of formal definitions, not of informal notion.

1 The Human Factor

A human whatever his occupation may be hardly has no personal view on the subject of the occupation. E. g. every researcher has probably his own concept of the continuum: either it is a set "composed" in a manner from all real numbers or something else — say, a kind of memory where all rational numbers are written and there are places in between for the other, irrational, numbers when they come into consideration.

If the researcher tries to formulate this concept then a description of a countable set arises — no other sets may be described. He may tell to his colleague the description and the latter says: "But this description is not full, since departing from it I may point out an object of the same kind differing from all objects falling under the description" (here lies the essence of the diagonal method which serves as a mean to prove many fundamental mathematical assertions). The reaction may be as sharp as "Nobody is interested in your object, in any case not me". This may be answered differently too, but there still remains the problem — to what extent the diagonal method is constructive and convincing or broader — the problem of a personal view on the science and its substances.

2 Abstract Computation

In the traditional computability theory (see, e. g., [2], ch. 5, other sources not quoted here contain analogous concepts and conclusions) a process of *computation* starts from some *input data* and ends in a favourable case with supplying of an appropriate *result*. An *abstract machine* is described which works over *data*. The process is usually divided in *steps*. On each step the machine executes just one *elementary action*, guided by a *rule* selected from a fixed finite collection. The *current* data are used — those available at the step beginning. The input data of the process serve as the current ones for the very first step.

A check is made too on each step whether the process is completed with no guarantee that it occurs at some time. Even a man observing the machine functioning (what is not forbidden but with no right to interfere) hardly if at all can in general case predict the future development of events.

The description of the sequence of rules which leads the machine to solving a *problem*, i. e. to getting a result tied in a specified manner with the input data, is called an *algorithm* of the problem solving. The algorithm may be considered as a *function* (in constructive sense of the word) usually built as a composition of a number of other functions.

Any abstract machine implements the idea of the *potential infinity.* Thus from any natural number n it is possible to pass to the number $n+1$. Regardless how many words in a finite alphabet are constructed there is a possibility to build a new word at any time.

The *recursion* applies the same idea to function computation. If the required result is not yet got then the function may — directly or via some other functions — call itself to continue the computation. The current recursion *level* is equal to the number of started and not yet terminated function evaluations. One specific function or all of them together may be accounted in the level. The recursion *depth* is equal to the maximum recursion level reached in the course of a function evaluation.

3 Is It Possible to Establish the Bound of the Recursion Depth?

This question occupies one of the leading places in the computability theory. Let us try to find a sufficiently general answer, limiting ourselves to the case of *end* recursion when the pass on the next level might take place only as the last step in the current level. To this goal we need two auxiliary functions without the parameters. The function

$B(\,) = \text{if } T \text{ then } T \text{ else } B(\,)$

breaks immediately from the loop of recursive calls with the value T ('true'), while the other one

$C(\,) = \text{if } T \text{ then } C(\,) \text{ else } T$

sticks in the loop forever.

Let us assume that a function $S(A, X)$ with two parameters: a function A and its argument X — may be described and that it solves the "stopping problem", i. e. supplies the value T if the evaluation of $A(X)$ ends successfully and the value F ('false') — otherwise. The traditional computability theory asserts that such an assumption leads to a contradiction.

The assertion has mainly the following proof. The function D is considered that predicts using S the result of the call $D(X)$. After that the computation is directed along the path consisting of either the call of B if the endless computation is predicted or the call of C — otherwise. In both cases the behaviour of either path and of the whole function contradicts to the prediction. Thus from the assumption on the function S the identically false result

$S(D,\, X) \equiv \neg S(D,\, X)$

may be deduced what leads to the conclusion that the function S with the required property cannot exist.

This proof would be perfect if the function D which makes exactly all said above might be described. Evidently the description should look like this:

$D(X) = \text{if } S(D,\, X) \text{ then } C(\,) \text{ else } B(\,)$

What is the value of the expression $S(D, X)$ occurring within it? Acting straightforwardly one may try to replace this expression by $D(X)$ (since the latter may have only T as its value). However this trial leads to no result at all, since such a form of D sticks already within the condition of the right part of the function definition and neither of the branches may be chosen. May the roundabout ways help?

4 The Static vs. Dynamic Approach to the Analysis of Algorithms

While building the function S the *static* approach is based only on the analysis of the text of the function A description. It is assumed that the value of S may be bound with the argument X value either without any knowledge of the latter or with very weak assumptions on its properties. The *dynamic* approach implies the analysis of the whole evaluation of $A(X)$ with the specific value of X. In the course of the process the properties of current data are looked through.

Let us look at the definitions of functions B and C given above. Each of them contains the call of function being defined. Therefore both may formally be considered recursive. However by a slightly more attentive look it occurs that one of two branches of the inner conditional expression is never chosen. It allows to bring these definitions to the form: $B(\,) = T$ and $C(\,) = C(\,)$, in which B ceases while C remains to be recursive. This is a rather trivial example of static analysis.

The notions of current data, current recursion level are intrinsically bound with the execution process and therefore should be considered dynamic. The notion of recursion depth is bound with the latter of them and seemingly should be taken dynamic too. However from the example of recursive definition of the function *factorial*:

$$\mathit{factorial}(N) = \text{if } N = 0 \text{ then } 1 \text{ else } N * \mathit{factorial}(N-1)$$

is rather evident that the recursion depth being equal to the value of the argument N (it may be easily proved by induction) may be sometimes found or estimated statically. But in the general case it is not so. E. g. the recursion depth of the easily defined function *hotpo*:

$$\begin{aligned}\mathit{hotpo}(N) = \text{if } N\ \mathit{mod}\ 2 = 0 \text{ then } \mathit{hotpo}(N \div 2)\\ \text{else if } N = 1 \text{ then } 1 \text{ else } \mathit{hotpo}(3 * N + 1)\end{aligned}$$

is very hard to estimate (I do not know whether someone has a luck to do it).

The question may be treated a bit more formally (but with the same result). The recursive function invariant (its least fixed-point) is the union of an infinite sequence of the continuing each other functions. Its k-th element A_k corresponds to the execution with the recursion depth k. If a finite form of the invariant is found then it provides the possibility of the static analysis. If however such a form is not found then the dynamical analysis becomes obligatory. It relies on the termination of the execution with the simultaneous obtaining both the result and its connection with the input data, i.e. in a sense on good luck.

The assumption that a function property may be always revealed statically should be declined since just this approach leads to the universally false assertion. The dynamic approach brings the computation of $S(D, X)$ into endless recursion: the function S calls D and that one again demands to find the value of $S(D, X)$. It is essential that this conclusion is obtained statically departing from the text of the function D description and from the choice of the dynamic approach to the function S implementation. The endless looping ruins the main idea of the diagonal method — to obtain the result with properties contradicting to predicted ones. As regards to the pair $\langle D, X\rangle$ the title question of section **3** may not be stated in a manner leading to any answer at all (the barber is bearded and the question — whether he shaves himself or not — looses any sense). Thus it is proved that the function S may be only partial: there exist functions (namely D) for which the static prediction of termination is impossible and the dynamic one falls in a deadlock because of peculiarities of the function structure.

So the diagonal method of reducing to a contradiction being so tempting in theory has not worked in practice. The discord between two colleagues in connection with the same method is coming to mind. However in books and papers it remains directly or not the leading method in proving assertions that algorithms of solving many mass problems are impossible. In that case such problems are called *algorithmically unsolvable*.

The guile of the intention — to compose the function D so that it behaves in spite of the prediction made by the function S — is not tightly bound with the situation. The places where B and C are called may be interchanged to make D to behave in accordance with the prediction. However that does not make the prediction possible.

5 A More General Case

Any function calling itself to bring a judgement whether it has some definite property is not shielded from endless looping. Let the algorithms either having or not having a property P do both exist and a function similar to D inherits the property from the chosen branch. The so called Rice (or Uspensky-Rice, see, e. g., [1], § 56) theorem states that no algorithm recognizing such a property is possible. In its proof the D-like function is used. The proof is as vulnerable as the previous one and on the same reason — the looping by the dynamic analysis of the couple $S + D$ behaviour arises inevitably before any prediction is made. This vulnerability lies on the surface being detected statically.

Only very seldom and for a very simple algorithms their properties may be found without their execution. Unfortunately such algorithms are typical for the publications on the "proof of programs correctness". In general case to determine that the result of an algorithm execution is bound in a certain manner with the input data is possible only while looking through and analyzing the algorithm action with a certain variant of the data.

Therefore instead of algorithmic *unsolvability* it is better to speak about algorithmic *semi-solvability* of almost all or of great many mass problems — the properties of the result are established together with its obtaining if this ever occurs.

6 Self-Applicability

In the traditional theory the first place among these problems occupies that of self-applicability of functions (see [1], § 47). A function is called either *self-applicable* or *self-inapplicable* depending on its applicability to its own description. The problem is stated: to build a function D, which is applicable only to the descriptions of all self-inapplicable functions (a variant of Russell's paradox). The trial is made: to prove using the diagonal method that the function building is impossible.

The assumption that D is self-applicable i. e. applicable to its description D' would mean in view of requirement to D that D' describes an self-inapplicable function. The arisen contradiction leads to the conclusion that D is self-inapplicable. The assumption that lack of self-applicability can be revealed statically again leads to contradiction. But by the dynamic approach no contradiction may arise. Indeed the absence of self-applicability of D means that one should infinitely long wait for the result of application of D to D', i. e. the second outcome of the prediction never will be available. In the same way the Russell's barber never meets the question — to shave or not to shave himself as a client.

7 The Freedom as a Realized Necessity

In [1], the remark to § 47.2.1, the non-admittance of a too large freedom in the context of set-theoretical conception is noted — it is impossible without falling in contradictions to combine freely any 'objects' in 'sets' which in turn are treated as 'objects'. However, on some reason there never arises the question whether without any limitation one may build 'words' — the descriptions of the 'algorithms' and to transfer these words on input of any algorithm.

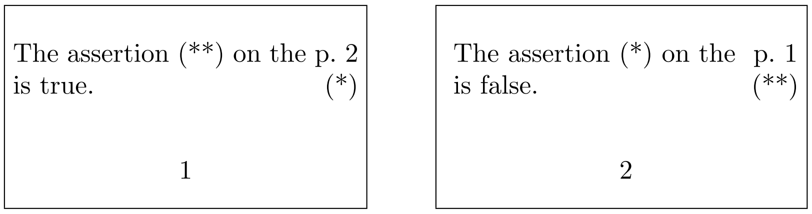
Maybe in the context of algorithm-theoretical approach the freedom may turn to be superfluous too? Let e. g. a normal algorithm is written assuming that its input word consists of two parts with a delimiter between them. Is there any reason to allow its application to the word containing no such delimiter? In other words — it is not reasonable to violate the simplest and well known to programmers limitation on types and to call a function of two parameters with only one argument.

Let us look from this point of view at the problem of self-applicability of the universal algorithm $\mathcal{U}$ ([1], § 48.2.1; [2], p. 248) transforming the pair $\langle \mathcal{A}, P \rangle$ into the result of applying an arbitrary algorithm $\mathcal{A}$ to a word P. What appearance the input word P must have in this case taking in mind the types?

Self-applicability demands this word to have the form $\langle \mathcal{U}, P_1 \rangle$. It will not be complete if P_1 differs from P. All this means that the word P must satisfy the equation $P = \langle \mathcal{U}, P \rangle$ obviously having no finite solution. This result may be considered as a static analogue of the assertion that the endless looping is inevitable in the attempt to prove dynamically using diagonal method that recognition the self-applicability of the universal algorithm is impossible.

In essence the "diagonal" function D and the like ones are endowed with the valuable meaning not in greater extent than verbal constructions used in the so called "semantic" paradoxes ([2], p. 9–10).

So one of the most expressive form of the "liar paradox" refers to the text two pages of which look like this:



The assertion (**) on the p. 2 is true. (*)

1

The assertion (*) on the p. 1 is false. (**)

2

It is generally accepted that paradoxes, which use references to other pages of the text, to labels like '(*)' at the reference place etc., lie out of the scope of the logic. However this motivation weakly agrees with the practice of mathematical publications.

It is also possible to admit that the assertions unable to be neither true nor false have no sense expected from the logical propositions.

But in solving this paradox it is necessary to take into account also that one of the assertions (*) and (**) was written before the other and therefore is the forecast of a future event. Whether such forecasts are acceptable in mathematics and if so have we a right to expect that they will be realized — that is the question deserving attention. The theory of approximation using the limited quantity of data is a field of mathematics, where the huge amount of conditions and counterexamples arises.

Let us return to the "diagonal" constructions. In a prophecy-free mathematics it is difficult to wait that an S-like function might be successfully applied to the analysis of algorithms appearing only after it was described.

It is generally accepted in physics that an observer may influence on the process observed — the presence of the observer makes the system not isolated. Something like we find in foundations of mathematics as well. By describing the field in which a mathematical theory is developed or a computational process takes place a person changes this field, brings a new element in it. Only catching this element he himself or somebody else can carry out the diagonal construction. As it was shown this construction concerns rather indirectly to the initial field.

8 Conclusion

In this paper written on the minimal level of formalization the next assertions are grounded:

- just this level is appropriate to the analysis of the problems taken from the so called foundations of mathematics;
- the contradiction grounding the impossibility of some algorithms arises only with the static approach;
- the dynamic approach leads only to the impossibility to judge definitely on these algorithms behaviour when the diagonal method is applied;
- some limitations generally accepted in set theory and in programming should be observed in computability theory as well, their violation leads to the unpredictable consequences;
- the researcher's influence on the results of considered problems need to be taken into account, especially in the case of the self-application of an algorithm.

References

1. A. A. Markov, N. M. Nagorny. *Algorithms theory* — Moscow, 1996 (in Russian)
2. E. Mendelson. *Introduction to mathematical logic* — Princeton etc.: van Nostrand Company, inc. (Russian translation: — Moscow, 1976)

Resolution and Binary Decision Diagrams Cannot Simulate Each Other Polynomially
Extended Abstract

Jan Friso Groote and Hans Zantema

Department of Computer Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`jfg@win.tue.nl`, `h.zantema@tue.nl`

Abstract. Proving formulas in propositional logic can be done in different ways. Some of these are based on of resolution, others on binary decision diagrams (BDDs). Experimental evidence suggests that BDDs and resolution based techniques are fundamentally different. This paper is an extended abstract of a paper [3] in which we confirm these findings by mathematical proof. We provide examples that are easy for BDDs and exponentially hard for any form of resolution, and vice versa, examples that are easy for resolution and exponentially hard for BDDs.

1 Introduction

We consider formulas in propositional logic: formulas consisting of proposition letters from some set $\mathcal{P}$, constants $\mathtt{t}$ (true) and $\mathtt{f}$ (false) and connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$ and $\leftrightarrow$. There are different ways of proving that a given formula is a tautology. In the automated reasoning community resolution is a popular proof technique, underlying the vast majority of all proof search techniques in this area.

In the VLSI and the process analysis communities binary decision diagrams (BDDs) are popular [2,5]. BDDs have caused a considerable increase of the scale of systems that can be verified, far beyond anything a resolution based method has achieved. On the other hand there are many examples where resolution based techniques out-perform BDDs with a major factor. Benchmarks showing out-performance in both directions have been described in [7].

However, these benchmark studies only provide an impression for the particular proof strategy used, saying very little about the real relation of resolution and BDDs. Actually, only given such benchmarks it can not be excluded that there exist a resolution based technique that always out-performs BDDs, provided a proper proof search strategy would be chosen. So, a mathematical comparison between the techniques is what we looked for and what we found.

Classical (polynomial) complexity bounds cannot be used, as the problem we are dealing with is (co-)NP-complete. Fortunately, polynomial simulations provide an elegant way of dealing with this (see e.g. [9]). We say that proof system A polynomially simulates proof system B if for every formula ϕ the size of the proof of ϕ in system A is smaller than a polynomial applied to the size of the proof of ϕ in system B. Of course, if the polynomial is more than linear, proofs in system A may still be substantially longer than proofs in system B, but at least the proofs in A are never exponentially longer. It is self evident that for practical applications it is important that the order of the polynomial is low. If it can be shown that for some formulas in B the proofs are exponentially longer than those in A we consider A as a strictly better proof system than B. It has for instance been shown that 'extended resolution' is strictly better than resolution [4], being strictly better than Davis-Putnam resolution; for an extended overview of comparisons of systems based on resolution, Frege systems and Gentzen systems we refer to [9].

As a main result we explicitly construct a sequence of contradictory formulas that are easy for BDDs, but exponentially hard for resolution. The proof that they are indeed hard for resolution is based on results from [8,1].

Conversely we explicitly construct a sequence of contradictory formulas that are easy for any reasonable form of resolution, even only unit resolution, but are exponentially hard for BDDs.

For proofs we refer to the full version [3] of this paper.

2 Binary Decision Diagrams

An Ordered Binary Decision Diagram (OBDD) is a Directed Acyclic Graph (DAG) where each node is labeled by a proposition letter from $\mathcal{P}$, except for nodes that are labeled by 0 and 1. From every node labeled by a proposition letter, there are two outgoing edges, labeled 'left' and 'right', to nodes labeled by 0 or 1, or a proposition letter strictly higher in some fixed ordering $<$ on $\mathcal{P}$. The nodes labeled by 0 and 1 do not have outgoing edges.

An OBDD compactly represents which valuations are valid, and which are not. Given a valuation σ and an OBDD B, the σ walk of B is determined by starting at the root of the DAG, and iteratively following the left edge if σ validates the label of the current node, and otherwise taking the right edge. If 0 is reached by a σ-walk then B makes σ invalid, and if 1 is reached then B makes σ valid. We say that an OBDD represents a formula if the formula and the OBDD validate exactly the same valuations.

An OBDD is called *reduced* if the following two requirements are satisfied.

- For no node do its left and right edge go to the same node.
- There are no two nodes with the same label of which the left edges go to the same node, and the right edges go to the same node.

The key property of reduced OBDDs states that for a fixed order $<$ on $\mathcal{P}$, every propositional formula ϕ is uniquely represented by a reduced OBDD $B(\phi)$, and ϕ and ψ are equivalent if and only if $B(\phi) = B(\psi)$.

As a consequence, a propositional formula ϕ is a contradiction if and only if $B(\phi) = 0$, and it is a tautology if and only if $B(\phi) = 1$. Hence by computing $B(\phi)$ for any suitable order $<$ we can establish whether ϕ is a contradiction, or

ϕ is a tautology, or ϕ is satisfiable. We write $\#(B(\phi))$ for the number of internal nodes in $B(\phi)$.

The main ingredient for the computation of $B(\phi)$ is the *apply*-operation: given the reduced OBBDs $B(\phi)$ and $B(\psi)$ for formulas ϕ and ψ and a binary connective $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ as parameters, the *apply*-operation computes $B(\phi \diamond \psi)$. For the usual implementation of *apply* as described in [2,5] both time and space complexity are $O(\#(B(\phi)) * \#(B(\psi)))$.

By the *OBDD proof* of a formula ϕ we mean the recursive computation of $B(\phi)$ using the *apply*-operation as described above. An alternative approach based on rewriting for computing reduced OBDDs has been given in [10].

3 Resolution

Contrary to the BDD technique, resolution is applied to formulas in *conjunctive normal form* (*CNF*), i.e. formulas of the form

$$\bigwedge_{i\in I} \bigvee_{j\in J_i} l_{ij}$$

where I and J_i are finite index sets and l_{ij} is a literal, i.e. a formula of the form p or $\neg p$ for a proposition letter p. Each sub-formula $\bigvee_{j\in J_i} l_{ij}$ is called a *clause.* As $\wedge$ and $\vee$ are associative, commutative and idempotent it is allowed and convenient to view clauses as sets of literals and CNFs as sets of clauses.

The resolution rule can be formulated by:

$$\frac{\{p, l_1, \ldots, l_n\} \quad \{\neg p, l'_1, \ldots, l'_{n'}\}}{\{l_1, \ldots, l_n, l'_1, \ldots, l'_{n'}\}}$$

where p is a proposition letter and l_i, l'_j are literals. A resolution proof of a set of clauses F is a sequence of clauses where the last clause is empty and each clause in the sequence is either taken from F, or matches the conclusion of the resolution rule, where both premises occur earlier in the sequence. Such a resolution sequence ending in the empty clause is called a *resolution refutation*, and proves that the conjunction of the set of clauses is a contradiction.

In case one of the clauses involved is a single literal l, by this resolution rule all occurrences of the negation of l in all other clauses may be removed. Moreover, all other clauses containing l then may be ignored. Eliminating all occurrences of l and its negation in this way is called *unit resolution.* All practical resolution proof search systems start with doing unit resolution as long as possible.

In order to apply resolution on arbitrary formulas, these formulas must first be translated to CNF. This can be done in linear time maintaining satisfiability using the Tseitin transformation [6]. A disadvantage of this transformation is the introduction of new variables, but without doing so transformation to CNF may be exponential.

The *Tseitin transformation* works as follows. Given a formula ϕ. Every sub-formula ψ of ϕ not being a proposition letter is assigned a new letter p_ψ. Now the Tseitin transformation of ϕ consists of

– the single literal p_ϕ;
– the conjunctive normal form of $p_\psi \leftrightarrow (p_{\psi_1} \diamond p_{\psi_2})$ for every subterm ψ of ϕ of the shape $\psi = \psi_1 \diamond \psi_2$ for a binary operator $\diamond$;
– the conjunctive normal form of $p_\psi \leftrightarrow \neg p_{\psi_1}$ for every subterm ψ of ϕ of the shape $\psi = \neg\psi_1$.

Here p_{ψ_i} is identified with ψ_i in case ψ_i is a proposition letter, for $i = 1, 2$. It is easy to see that this set of clauses is satisfiable if and only if ϕ is satisfiable. Moreover, every clause consists of at most three literals, and the number of clauses is linear in the size of the original formula ϕ.

By a resolution proof for an arbitrary formula we mean a resolution proof after the Tseitin transformation has been applied.

4 The Main Result

First we construct formulas that are hard for OBDD proofs but easy for resolution.

For n a positive integer and p_{ij} distinct variables we define

$$CR_n = (\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{n} p_{ij})) \wedge (\bigwedge_{j=1}^{n} (\bigvee_{i=1}^{n} p_{ij})).$$

Imagine the variables in a matrix according to the indexes, then CR_n states that in every column and in every row of the matrix at least one variable is true.

For every order $<$ on $\mathcal{P}$ it turns out that $\#B(CR_n) = \Omega(1.63^n)$. As a consequence we arrive the same lower bound for any OBDD proof of the contradictory formula $p \wedge (\neg p \wedge CR_n)$ since computation of $B(CR_n)$ is part of this OBDD proof.

Conversely it is not difficult to prove that applying only unit resolution on the Tseitin transformation of this formula yields a refutation in a number of steps linear in the size of the formula. Hence:

Theorem 1 *For the contradictory formulas $p \wedge (\neg p \wedge CR_i)$ of size $\Theta(i^2)$ $(i \geq 0)$ yielding a refutation in $O(i^2)$ steps using only unit resolution, every OBDD proof has time and space complexity $\Omega(1.63^i)$.*

Next we construct formulas that are easy for OBDD proofs but hard for resolution.

For a string $S = p_1, p_2, p_3, \ldots, p_n$ of proposition letters, where letters are allowed to occur more than once, we write

$$[S] \ = \ p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \cdots (p_{n-1} \leftrightarrow p_n)) \cdots).$$

It is not difficult to see that $[S]$ is a tautology if and only if all letters occur an even number of times in S.

Since all subformulas of a formula composed only of $\leftrightarrow$, $\neg$ and proposition letters turn out to have a linear size reduced OBDD, any OBDD proof of such a formula is polynomial. More precisely, the complexity of the OBDD proof of $\neg[S]$ is $O(|S|^2)$.

We now give a construction of strings S_n in which all letters occur exactly twice by which $\neg[S_n]$ is a contradiction, and for which every resolution proof of $\neg[S_n]$ is very long.

For a string S and a label i we write $\mathsf{lab}(S, i)$ for the string obtained from S by replacing every symbol p by a fresh symbol p_i. For a string S of length $n * 2^n$ we write $\mathsf{ins}(n, S)$ for the string obtained from S by inserting the symbol i after the $(i * n)$-th symbol for $i = 1, 2, \ldots, n$. We define

$$S_1 = 1, 1, \quad \text{and}$$

$$S_{n+1} = \mathsf{ins}(n, \mathsf{lab}(S_n, 0)), \mathsf{ins}(n, \mathsf{lab}(S_n, 1)),$$

for $n > 0$. For instance, we have

$$S_1 = \underbrace{1}, \underbrace{1},$$

$$S_2 = \underbrace{1_0, 1}, \underbrace{1_0, 2}, \underbrace{1_1, 1}, \underbrace{1_1, 2},$$

$$S_3 = \underbrace{1_{00}, 1_0, 1}, \underbrace{1_{00}, 2_0, 2}, \underbrace{1_{10}, 1_0, 3}, \underbrace{1_{10}, 2_0, 4}, \underbrace{1_{01}, 1_1, 1}, \underbrace{1_{01}, 2_1, 2}, \underbrace{1_{11}, 1_1, 3}, \underbrace{1_{11}, 2_1, 4}.$$

Clearly S_n is a string of length $n * 2^n$ over $n * 2^{n-1}$ symbols each occurring exactly twice. The string S_n is constructed to consist of 2^n consecutive groups of n symbols; in the examples S_1, S_2 and S_3 above these groups are under-braced.

Using results from [8,9,1] it turns out that every resolution proof of $\neg[S_n]$ contains $2^{\Omega(2^n/n)}$ resolution steps. In order to be able to formulate this result without double exponentiation define for every i the formula ϕ_i to be $\neg[S_n]$, where n is the smallest number satisfying $i \leq \frac{2^n}{n}$. Hence:

Theorem 2 *For the contradictory formulas ϕ_i of size $\Theta(i \log^2 i)$ $(i \geq 0)$ every OBDD proof has time and space complexity $O(i^2 \log^4 i)$, and every resolution proof requires $2^{\Omega(i)}$ resolution steps.*

References

1. Ben-Sasson, E., and Wigderson, A. Short proofs are narrow – resolution made simple. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing* (1999), pp. 517–526.
2. Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (1986), 677–691.
3. Groote, J. F., and Zantema, H. Resolution and binary decision diagrams cannot simulate each other polynomially. *Journal of Discrete Applied Mathematics* (2001). To appear.
4. Haken, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297–308.
5. Meinel, C., and Theobald, T. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications.* Springer, 1998.

6. Tseitin, G. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, part 2* (1968), pp. 115–125. Reprinted in J. Siekmann and G. Wrightson (editors),*Automation of reasoning* vol. 2, pp. 466-483.,Springer–Verlag Berlin, 1983.
7. Uribe, T. E., and Stickel, M. E. Ordered binary decision diagrams and the Davis-Putnam procedure. In *First conference on Constraints in Computational Logic* (1994), J.-P. Jouannaud, Ed., vol. 845 of *Lecture Notes in Computer Science*, Springer, pp. 34–49.
8. Urquhart, A. Hard examples for resolution. *Journal of the ACM 34*, 1 (1987), 209–219.
9. Urquhart, A. The complexity of propositional proofs. *The Bulletin of Symbolic Logic 1*, 4 (1995), 425–467.
10. Zantema, H., and van de Pol, J. C. A rewriting approach to binary decision diagrams. *Journal of Logic and Algebraic Programming* (2001). To appear.

On Expressive and Model Checking Power of Propositional Program Logics

Nikolai V. Shilov[1,2] and Kwang Yi[1]

[1] Korean Advanced Institute of Science and Technology, Taejon 305-701, Kusong-dong Yusong-gu 373-1, Korea (Republic of Korea), {shilov, kwang}@ropas.kaist.ac.kr

[2] A. P. Ershov Institute of Informatics Systems Russian Academy of Sciences, Siberian Branch 6, Lavrent'ev ave., 630090, Novosibirsk, Russia shilov@iis.nsk.su

Abstract. We examine when a model checker for a propositional program logic can be used for checking another propositional program logic in spite of lack of expressive power of the first logic. We prove that (1) a branching time Computation Tree Logic CTL, (2) the propositional μ-Calculus of D. Kozen μC, and (3) the second-order propositional program logic 2M of C. Stirling enjoy the equal model checking power in spite of difference in their expressive powers CTL $<$ μC $<$ 2M: every listed logic has a formula such that every model checker for this particular formula for models in a class closed w.r.t. finite models, Cartesian products and power-sets can be reused for checking all formulae of these logics in all models in this class. We also suggest a new second-order propositional program logic SOEPDL and demonstrate that this logic is more expressive than 2M, is as expressive as the Second order Logic of monadic Successors of M. Rabin (**S(n)S**-Logic), but still enjoys equal model checking power with CTL, μC and 2M (in the same settings as above).

1 CTL and μC vs. Second-Order Logics

The propositional μ-Calculus of D. Kozen (μC) [10,11] is a powerful propositional program logic with fixpoints. In particular, a very popular with model checking community state-based temporal Computation Tree Logic (CTL) [7, 4,5] is interpretable in μC. It is almost a folklore that CTL is less expressive than μC. Nevertheless, expressibility problem rises every time when a particular property is concerned: whether this property can be expressed in CTL, in μC, or both logics are inadequate.

For example, paper [1] reports a progress in experiments with symbolic data representation by means of Binary Decision Diagrams (BDD) for solving some board games. In this research positions and moves are represented by BDD, an existence of winning strategy is expressed as a simple μC formula. Then a new model checking tool for a fragment of μC and finite models presented by

BDDs has been applied. A natural question rises: why these experiments have exploited a new symbolic model checking tool instead of utilizing a very popular and reliable symbolic model checker for CTL? A natural move for exploiting a model checker of this kind for solving board games is to try to express an existence of winning strategy in finite games in terms of CTL formulae and then to experiment with these formulae, games presented by BDDs and the model checker. But this move is doomed to fail as follows from the game-theoretic arguments below.

There are different formalisms for games. We would like the following: a game (of two plays A and B) (with terminal positions) is tuple $(P_A, P_B, M_A, M_B, W_A, W_B)$, where

– $P_A \cap P_B = \emptyset$ are nonempty sets of positions,
– $M_A \subseteq (P_A \setminus (W_A \cup W_B)) \times (P_A \cup P_B)$, $M_B \subseteq (P_B \setminus (W_A \cup W_B)) \times (P_A \cup P_B)$ are moves of A and B respectively,
– $W_A, W_B \subseteq (P_A \cup P_B)$ are sets of winning positions for A and B.

The game is said to be finite iff P_A and P_B are finite. A session of the game is a maximal sequence of positions $s_0,\ ...s_n,\ ...$ such that $(s_i, s_{i+1}) \in (M_A \cup M_B)$ for all consequentive positions $s_i, s_{i+1}\ \in (s_0,\ ...s_n,\ ...)$. A player $C \in \{A, B\}$ wins a session iff the session finishes in F_C. A strategy of a player is a subset of the player's possible moves. A winning strategy for a player is a strategy of the player which always leads to the player's win: the player wins every session in which he/she implements this strategy instead of all moves.

Proposition 1.
1. No CTL formula can express existence of winning strategies in finite games.
2. μC formula $\mu\ x.\ (P_A \vee \langle M_A \rangle x \vee (\langle M_B \rangle true \wedge [M_B]x))$ expresses existence of winning strategies for player A against B in games with terminal positions.

Proof. We would like to prove the first part only. We can consider CTL formulae without constructions **AG**, **AF**, **EG**, and **EF** at all due to the following equivalences:

$$\begin{array}{ll} \mathbf{AF}\phi \ \leftrightarrow \mathbf{A}(true\mathbf{U}\phi) & \mathbf{EG}\phi \ \leftrightarrow \neg\mathbf{AF}(\neg\phi) \\ \mathbf{EF}\phi \ \leftrightarrow \mathbf{E}(true\mathbf{U}\phi) & \mathbf{AG}\phi \ \leftrightarrow \neg\mathbf{EF}(\neg\phi) \end{array}$$

Let us define nesting for CTL formulae of this kind as follows:
$nest(true) = nest(false) = 0$,
$nest(r) = 0$ for every propositional variable r,
$nest(\neg\phi) = nest(\phi)$,
$nest(\phi \wedge \psi) = nest(\phi \vee \psi) = \max\{nest(\phi), nest(\psi)\}$,
$nest(\mathbf{A}(\phi\mathbf{U}\psi)) = nest(\mathbf{E}(\phi\mathbf{U}\psi)) = \max\{nest(\phi), nest(\psi)\}$,
$nest(\mathbf{AX}\phi) = nest(\mathbf{EX}\phi) = 1 + nest(\phi)$.
Then let us consider a game NEG (Fig. 1), where A/B-line represents positions where a player A/B has moves, downward/upward arrows represent moves of A/B, W_A/W_B represents a single winning position for A/B. Then $(-k_A) \models_{NEG} \phi \Leftrightarrow (-l_B) \models_{NEG} \phi$ holds for every CTL formula ϕ and for all $k, l > nest(\phi)$.

$$\begin{tikzcd}[column sep=small]
 & & \overbrace{\hspace{6em}}^{NEG_i} & & \\
 & & & & W_A \\
A: \ldots (-i-1) \arrow[drr] & (-i) \arrow[dr] & \ldots & (-1) \arrow[dr] & (0) \\
\ldots & & & & W_B \\
B: \ldots (-i-1) \arrow[urr] & (-i) \arrow[ur] & \ldots & (-1) \arrow[ur] & (0) \\
 & & \underbrace{\hspace{6em}}_{NEG_i} & &
\end{tikzcd}$$

Fig. 1. Model NEG and NEG_i

Hence for every formula of CTL there exists a non-positive number prior to which the formula is a boolean constant in the NEG. Let us also remark that no CTL formula can distinguish positions of a finite game NEG_i (fig. 1) from correspondent positions of NEG. But A has a winning strategy in all even integers on A-line and in all odd integers on B-line. Hence no CTL formula can express an existence of a winning strategy for a player A in positions of finite games NEG_i for $i \geq 0$. Thus the proof of proposition 1 is **over.**

Another evidence of μC expressive power comes from comparison with the Second-order logic of several monadic Successors of M. Rabin (**S(n)S**-Logic) [12, 13,2]: both logics enjoy equal expressive power on infinite trees [14]. But in spite of this, μC fails to express some very natural properties discussed below. For resolving the problem with a deficit of expressive power of μC, the second order propositional program logic 2M of C. Stirling [17] uses second order quantifiers $\forall/\exists$ over propositional variables and reachability modalities $\Box/\Diamond$ upon strongly connected components of models.

Proposition 2.
1. No μC formula can express commutativeness of composition of action symbols in finite models.
2. 2M formula $\forall p.\big(([a][b]p) \leftrightarrow ([b][a]p)\big)$ expresses commutativeness of composition of action symbols a and b in models.

But some other very natural and useful properties are still inexpressible in 2M. For instance 2M can not express the weakest precondition for inverse actions as it is defined below. Assume that a propositional variable p is interpreted in a model by a set of states P, and an action symbol r is interpreted by a binary relation R on states. The weakest pre-condition for inverse of r with respect to a post-condition p in this model is the following state of states $\{t : \forall\ s.((s,t) \in R \Rightarrow s \in P)\}$. We would like to suggest a another Second-Order Elementary Propositional Dynamic Logic (SOEPDL) for handling this phenomenon. A single difference between 2M and SOEPDL is interpretation of reachability modalities $\Box/\Diamond$: in SOEPDL they means "for every/some state in the model".

Proposition 3.
1. No 2M formula can express in finite models the weakest pre-conditions for inverse of action symbol with respect to propositional variable.
2. SOEPDL formula $\exists q.\big(\Box(\langle a\rangle q \to p) \wedge q\big)$ expresses the weakest pre-conditions for inverse of action symbol a with respect to propositional variable p in models.

2 Expressiveness vs. Model Checking

Theorem 1.
CTL $<$ μC $<$ 2M $<$ SOEPDL where all expressibilities have linear complexity and all inexpressibilities can be justified in finite models.

Proof.
First, CTL $<$ μC since it is well known that CTL $\leq$ μC and, in accordance with proposition 1, there is μC formula, which is not equivalent to any CTL formula in finite models.

Next, μC $<$ 2M since $(\mu p.\phi) \leftrightarrow (\forall p.(\Box(\phi \to p) \to p))$, $(\nu p.\phi) \leftrightarrow (\exists p.(\Box(p \to \phi) \wedge p))$, and, in accordance with proposition 2, there is 2M formula, which is not equivalent to any μC formula in finite models.

Finally, 2M $<$ SOEPDL since reachability modalities can be expressed in terms of PDL programs as $\Box\phi \leftrightarrow [(\cup_{a\in Act} a)*]\phi$ and $\Diamond\phi \leftrightarrow \langle(\cup_{a\in Act} a)*\rangle\phi$, and, in accordance with proposition 3, there is SOEPDL formula, which is not equivalent to any 2M formula in finite models.

Thus the proof of the theorem 1 is **over.**

An "internal" characteristic of the expressive power of SOEPDL in terms of other propositional program logics correlates with an "external" one: we demonstrate that SOEPDL is as expressive as Second Order Logic of **n** Monadic Successors of M. Rabin (**S(n)S**-Logic) [12,13,2]. We would not like to give a complete definition of **S(n)S**-Logic, but would like to point out that we use action symbols as symbols of monadic functions (i.e., "successors") and exploit functional models, where action symbols are interpreted as total monadic functions. In these settings functional models are just special kind of SOEPDL models. Boolean values of formulae of **S(n)S**-Logic in functional models depend on values of free individual variables only. Thus boolean values of formulae of **S(n)S**-Logic with a single (at most) free individual variable depend on values of this single variable. In this setting semantics of formulae of **S(n)S**-Logic with a single (at most) free first order variable is a state-based semantics as well as semantics of SOEPDL and it is possible to compare semantics of formulae of **S(n)S**-Logic of this kind and SOEPDL formulae in terms of equivalence.

Theorem 2. *Expressive powers of SOEPDL and* **S(n)S**-*Logic are linear time equivalent in the following sense:*

- *for every formula of* **S(n)S**-*Logic with a single (at most) free first-order variable it is possible construct in linear time an equivalent in all functional models formula of SOEPDL;*
- *for every formula of SOEPDL it is possible construct in linear time an equivalent in all functional models formula of* **S(n)S**-*Logic with a single (at most) free first-order variable.*

Expressiveness is a particular dimension where we can compare a power of program logics. Another possible dimension is model checking power. Let us discuss it below. Global (model) checking consists in a calculation of the set

– $(s, S_x, ...S_z, (\psi_1 \substack{\wedge \\ \vee} \psi_2)) \to (s, S_x, ...S_z, \psi_i)$ for every $i \in \{1, 2\}$;
– $(s, S_x, ...S_z, (\frac{[a]}{\langle a \rangle}\psi)) \to (t, S_x, ...S_z, \psi)$ for every t such that $(s, t) \in I(a)$;
– $(s, S_x, ...S_z, (\substack{\Box \\ \Diamond}\psi)) \to (t, S_x, ...S_z, \psi)$ for every t in M;
– $(s, S_x, ...S_y, ...S_z, (\substack{\forall \\ \exists} y.\psi)) \to (s, S_x, ...S, ...S_z, \psi)$ for every $S \subseteq D$.

Fig. 2. Moves of Falsifier/Verifier

of all states of an input model where an input formula is valid. Local (model) checking is checking an input formula in an input state of an input model. If LG is a logic and MD is a class of models, then a model checker for LG×MD is a program (algorithm) which can check all LG formulae in all MD models. Assume LG′ is a propositional program logic and MC′ be a model checker for LG′×MD. Assume we would like to check formulae of another propositional program logic LG″ in MD models. A first move is to try to reuse MC′, i.e., to force MC′ to do this job instead of expensive and risky design, implementation and validation of a new model checker MC″ for LG″×MD. If LG″ $\leq$ LG′ then the work is done. The question is: when LG″ $\not\leq$ LG′, is it still possible to reuse MC′ for LG″×MD? In particular, whether a model checker of CTL formulae in finite models can be utilized for solving board games in spite of lack of expressive power of CTL?

Let ξ be SOEPDL formula. Without loss of generality we can assume that variables bounded by different quantifiers are different (so there are no name collisions). Moreover we can assume that negations in the formula are applied to propositional variables only since the following equivalences hold:
$(\neg(\neg\phi)) \leftrightarrow \phi$
$(\neg(\phi \wedge \psi)) \leftrightarrow ((\neg\phi) \vee (\neg\psi))$ $(\neg(\langle a \rangle\phi)) \leftrightarrow ([a](\neg\phi))$
$(\neg(\Diamond\phi)) \leftrightarrow (\Box(\neg\phi))$ $(\neg(\exists p.\phi)) \leftrightarrow (\forall p.(\neg\phi))$
Let x, ... z be a list of all bounded propositional variables in ϕ. A subformula of ξ is a formula, which is (syntactically) a part of ϕ. A subformula is said to be *conjunctive* iff it has one of the following forms: $\phi \wedge \psi$, $[a]\phi$, $\Box\phi$ or $\forall p.\phi$. A subformula is said to be *disjunctive* iff it has one of the following forms: $\phi \vee \psi$, $\langle a \rangle\phi$, $\Diamond\phi$ or $\exists p.\phi$. Propositional variables and their negations (and only they) are neither conjunctive nor disjunctive subformulae. Let M be a finite model with a domain D and an interpretation I.

We are going to construct a Hintica-like finite game $G(M, \xi)$ of two players *Falsifier* and *Verifier*. Positions in this game $G(M, \xi)$ are tuples $(s, S_x, ...S_z, \psi)$ where $s \in D$ is a current state, $S_x, ...S_z \subseteq D$ are current interpretations for x, ... y, and ψ is a verified subformula of ξ. Falsifier/Verifier has moves of 4 kinds (Fig. 2) related conjunctive/disjunctive subformulae respectively, and wins in positions of 6 kinds (Fig. 3). Intuitively: Verifier is trying to validate a formula in a state while a rival Falsifier is trying to falsify the formula in a state.

The following is easy to prove by induction on formula structure.

Proposition 4. *For every finite model $M = (D, I)$ and every SOEPDL formula ξ there exists a finite game $G(M, \xi)$ of two players Falsifier and Verifier such that the following holds for every state $s \in D$ and every subformula ϕ of ξ:*

- $(s, S_x, ...S_z, \frac{false}{true})$,
- $(s, S_x, ...S_z, p)$, where p is a free propositional variable and $s\genfrac{}{}{0pt}{}{\notin}{\in}I(p)$,
- $(s, S_x, ...S_z, \neg p)$, where p is a free propositional variable and $s\genfrac{}{}{0pt}{}{\in}{\notin}I(p)$,
- $(s, S_x, ...S_z, (\frac{\langle a\rangle}{[a]}\psi))$ iff $(s,t) \notin I(a)$ for every t;
- $(s, S_x, ...S_y, ...S_z, y)$, where $s\genfrac{}{}{0pt}{}{\notin}{\in}S_y$,
- $(s, S_x, ...S_y, ...S_z, \neg y)$, where $s\genfrac{}{}{0pt}{}{\in}{\notin}S_y$.

Fig. 3. Winning positions of Falsifier/Verifier

Verifier has a winning strategy against Falsifier in a position $(s, I(x), ...I(z), \phi)$ iff $s \models_M \phi$ (where $x, ...z$ is a list of all bounded variables of ξ). The game can be constructed in time $O(2^{d\times f} \times d \times f)$, where d is amount of states and f is formula size.

It immediately implies the following sufficient

Criterion *Let L' be a program logic and MC be a model checker for this logic L' in finite models which can check an existence of a winning strategy for a player against a counterpart in positions of finite games with time complexity $T(m)$, where m is an overall game size. Let L'' be another logic which is expressively equivalent to a fragment of SOEPDL and $C(f)$ and $S(f)$ be time and space complexity of a translation L'' formulae into SOEPDL formulae, where f is a formula size. Then MC can be reused for checking L'' formulae in finite models and upper time bound for model checking L'' in finite models is $max\{C(f),\ T(O(2^{d\times S(f)} \times d \times S(f)))\}$, where d is amount of states in the model.*

A consequence of the above criterion, proposition 1 and theorem 1 is the following

Theorem 3. *Let MC be a model checker which implements linear in an overall model size model checking algorithm for the following formula $\mu\ x.\ (p \vee \langle a\rangle x \vee (\langle b\rangle true \wedge [b]x))$ of μC in finite models. MC can be reused for checking all formulae of SOEPDL, 2M, and μC in finite models with upper time bound $exp(d \times f)$, where d is amount of states in a model and f is formula size.*

We would like to remark that a formula $\mu\ x.\ (p \vee \langle a\rangle x \vee (\langle b\rangle true \wedge [b]x))$ from the theorem above can be checked in finite models in linear time [6].

We would like to generalize the above theorem 3 as follows below. A class of models MD is closed with respect to Cartesian products iff for all models M' and M'' in MD, every model M with $D_M = D_{M'} \times D_{M''}$ is in MD also. A class of models MD is *closed with respect to power-sets* iff for every model M' in MD, every model M with $D_M = \mathcal{P}(D_{M'})$ is in MD also. Due to space limitations we would like to present the following theorem without proof.

Theorem 4. *Let MD be a class of models which contains all finite models and is closed with respect to Cartesian products and power-sets. Let MC be a model checker which can check (at least) the following formula ($\mathbf{EF}p$) of CTL in models in MD. Then MC can be reused for checking all formulae of SOEPDL, 2M, μC and CTL in models in MD.*

3 Conclusion

We began this research from a particular problem whether a model checker for CTL formulae in finite models can be used for solving board games. Now we content our curiosity: in principle yes it is (theorem 4), in spite of lack of expressive power (proposition 1). Theoretical method suggested for it is very simple from algebraic viewpoint, since it exploits Cartesian products and power set operation. But it is too expensive in computation complexity and impractical due to double exponential blow up of a model (power set operation is used twice).

In general, in spite of algorithmical inefficiency of presented results, contribution of this paper is two-folds. First we study expressive and model checking power of the classical propositional logic or a basic propositional modal logic **K** [3,15], extended by transitive closure (CTL), fixpoints (μC), and second-order monadic quantification (2M and SOEPDL). We consider this study as a propositional counterpart of a popular research topic in descriptive complexity, where expressive power and finite spectra are examining for First-Order Logic, extended by transitive closure (FOL+TC), fixpoints (FOL+FP), and second-order quantification (SOL) [9].

Next contribution consists in model checkers reuse. Let us discuss it with some details in the next paragraph. Assume we have a reliable model checker MC for CTL in finite models, but we would like to check in finite models formulae of more powerful combined logic (CTL+PDL) extended by program intersection $\cap$ (it is essential for representation and reasoning about *distributed* knowledge in logic of knowledge [8]). This extended logic can not be expressed neither in CTL nor in μC (due to presence of program intersection), but in SOEPDL only as follows: $\forall p.([\alpha \cap \beta]p \leftrightarrow ([\alpha]p \wedge [\beta]p))$. Does it imply a necessity to implement a new model checker in stead of MC? Or does it imply that code revision and patching of MC are inevitable? — Not at all, as it follows from equal model checking power of CTL and SOEPDL! One can try to encode extensions in models instead of risky implementation and patching. If it can be done in feasible time and space in terms of MC's input language for models, then just implement it as a preprocessor for MC, and reuse MC for extended logic. In particular, this approach is valid for (CTL+PDL) extended by program intersection $\cap$, since an encoding in this particular case is linear in time and space. But in general, better analysis when model checkers for CTL and μC in finite models can be reused for model checking other logics in finite models without loss of efficiency is a research topic for a perspective.

Finally we would like to remark that close connections between model checking for μC and special games have been extensively examined [16,17,18]. In particular, [16] has defined infinite model checking games and established an equivalence of local model checking to an existence of winning strategies. Then [17] has defined *finite* fixed point games and characterized indistinguishability of processes by means of formulae with bounded amounts of modalities and fixpoints in terms of winning strategies with bounded amounts of moves. Paper [18] has exploited model-checking games for practical efficient local model checking. Paper [17] has defined also logic 2M, corresponding games and established indistinguishability of states by means of formulae with bounded amounts of modalities and quantifiers in terms of winning strategies with bounded amounts of moves. So it is possible to summarize that [16,17,18] have exploited infinite games for local model checking μC in infinite models. In contrast, we exploit games with terminal positions (basically, finite games) for forcing model checkers for CTL to check more expressive logics μC, 2M, and SOEPDL.

References

1. Baldamus M, Schneider K., Wenz M., Ziller R. *Can American Checkers be Solved by Means of Symbolic Model Checking?* Electronic Notes in Theoretical Computer Science, v.43, `http://www.elsevier.nl/gej-ng/31/29/23/show/Products/notes/`
2. Börger E., Grädel E., Gurevich Y. *The Classical Decision Problem.* Springer, 1997.
3. Bull R.A., Segerberg K. *Basic Modal Logic.* Handbook of Philosophical Logic, v.II, Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.), p.1-88.
4. Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J. *Symbolic Model Checking: 10^{20} states and beyond.* Information and Computation, v.98, n.2, 1992, p.142-170.
5. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.
6. Cleaveland R., Klain M., Steffen B. *Faster Model-Checking for Mu-Calculus.* Lecture Notes in Computer Science, v.663, 1993, p.410-422.
7. Emerson E.A. *Temporal and Modal Logic. Handbook of Theoretical Computer Science*, v.B, Elsevier and The MIT Press, 1990, 995-1072.
8. Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995.
9. Immerman N *Descriptive Complexity: a Logician's Approach to Computation.* Notices of the American Mathematical Society, v.42, n.10, p.1127-1133.
10. Kozen D. *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.
11. Kozen D., Tiuryn J. *Logics of Programs.* Handbook of Theoretical Computer Science, v.B, Elsevier and The MIT Press, 1990, 789-840.
12. Rabin M.O. *Decidability of second order theories and automata on infinite trees.* Trans. Amer. Math. Soc., v.141, 1969, p.1-35.
13. Rabin M.O. *Decidable Theories.* in Handbook of Mathematical Logic, ed. Barwise J. and Keisler H.J., North-Holland Pub. Co., 1977, 595-630.
14. Schlingloff H. *On expressive power of Modal Logic on Trees.* LNCS, v.620, 1992, p.441-450.
15. Stirling C. *Modal and Temporal Logics.* Handbook of Logic in Computer Science, v.2, Claredon Press, 1992, p.477-563.
16. Stirling C. *Local Model Checking Games.* Lecture Notes in Computer Science, v.962, 1995, p.1-11.
17. Stirling C. *Games and Modal Mu-Calculus.* Lecture Notes in Computer Science, v.1055, 1996, p.298-312.
18. Steven P., Stirling C. *Practical Model Checking Using Games.* Lecture Notes in Computer Science, v.1384, 1998, p.85-101.

An Extension of Dynamic Logic for Modelling OCL's @*pre* Operator

Thomas Baar, Bernhard Beckert, and Peter H. Schmitt

Universität Karlsruhe, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
Fax: +49 721 608 4211, {baar,beckert,pschmitt}@ira.uka.de

Abstract. We consider first-order Dynamic Logic (DL) with non-rigid functions, which can be used to model certain features of programming languages such as array variables and object attributes. We extend this logic by introducing, for each non-rigid function symbol f, a new function symbol $f^{@pre}$ that after program execution refers to the old value of f before program execution. We show that DL formulas with @*pre* can be transformed into equivalent formulas without @*pre*. We briefly describe the motivation for this extension coming from a related concept in the Object Constraint Language (OCL).

1 Introduction

Since the Unified Modeling Language (UML) [11] has been adopted as a standard of the Object Management Group (OMG) in 1997, many efforts have been made to underpin the UML — and the Object Constraint Language (OCL), which is an integral part of the UML, — with a formal semantics. Most approaches are based on providing a translation of UML/OCL into a language with a well-understood semantics, e.g., BOTL [5] and the Larch Shared Language (LSL) [6].

Within the KeY project [1], we follow the same line, translating UML/OCL into Dynamic Logic (DL).[1] This choice is motivated by the fact that DL can cope with both the dynamic concepts of UML/OCL and real world programming languages used to implement UML models (e.g. Java Card [4]).

The OCL allows a UML model to be enriched with additional constraints, e.g., invariants for UML classes, pre-/post-conditions for operations, guards for transitions in state-transition diagrams, etc. Although, at first glance, OCL is similar to an ordinary first-order language, closer inspection reveals some unusual concepts. The @*pre* operator is among them. In OCL, this unary operator is applicable to attributes, associations, and side-effect-free operations (these are called "properties" in the OCL context [11, p. 7-11ff]). The @*pre* operator can only be used in post-conditions of UML operations. A property *prop* followed by @*pre* in the post-condition of an operation $m()$ evaluates to the value of *prop* before the execution of $m()$.

[1] More information on the KeY project can be found at `i12www.ira.uka.de/~key`.

Dynamic Logic [7,8,9,10] can be seen as an extension of Hoare logic [2]. It is a first-order modal logic with modalities $[p]$ and $\langle p \rangle$ for every program p. The models used to define the semantics of DL consist of the possible program states. The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p when started in the current state, and $\langle p \rangle \phi$ expresses that ϕ holds in *at least one* of them. In versions of DL with a non-deterministic programming language there can be several such final states. For deterministic programs there is exactly one final state (if p terminates) or there is no final state (if p does not terminate). In that case, the formula $\phi \to \langle p \rangle \psi$ is valid if, for every state s satisfying pre-condition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds. The formula $\phi \to [p]\psi$ expresses the same, except that termination of p is not required, i.e., ψ must only hold *if* p terminates. Thus, $\phi \to [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$.

Here, we consider a version of first-order DL with non-rigid functions, i.e., functions whose interpretation can differ from state to state. Non-rigid functions can be used to model features of real-world programming languages such as array variables and object attributes.

Moreover, to ease the translation of OCL into DL, we extend DL by introducing, for each non-rigid function symbol f, a new corresponding function symbol $f^{@pre}$ that has a specially tailored semantics. The new symbol $f^{@pre}$ refers in the final state of a program execution to the value of the corresponding symbol f in the initial state of the program execution. This allows to easily express the relation between the initial and the final state. For example, $[p](c \doteq c^{@pre})$ expresses that the interpretation of the constant c is not changed by the program p.

In Section 2, we briefly introduce DL with non-rigid functions. In Section 3 we add function symbols with $@pre$ to DL and formally give their semantics. Finally, in Section 4, we present two transformations from DL with $@pre$ into DL without $@pre$.

The main contribution of this paper is to show that DL formulas containing the new function symbols with $@pre$ can be transformed into equivalent formulas without them. An extended version of this paper containing additional material, including examples and proofs, is available [3].

2 Dynamic Logic with Non-rigid Functions

Although non-rigid functions are mostly ignored in the literature, the more specific concept of array assignments has been investigated [7,9]. In both papers their semantics is handled by adding to each state valuations of second-order array variables. We introduce, instead, non-rigid function symbols. This shift of attention comes naturally when we want to axiomatise the semantics of object-oriented languages in DL. In this setting non-static attributes of a class are best modelled by non-rigid functions.

Let $\Sigma = \Sigma_{nr} \cup \Sigma_r$ be a signature, where Σ_{nr} contains the non-rigid function symbols and Σ_r contains the rigid function symbols and the predicate symbols, which are all rigid (Σ_r always contains the equality relation $\doteq$). The set $Term(\Sigma)$

of terms and the set $Atom(\Sigma)$ of atomic formulas are built as usual from Σ and an infinite set Var of object variables. A term is called *non-rigid* if (a) it is a variable or (b) its leading function symbol is in Σ_{nr}.

The programs in our DL are (deterministic) *while* programs with a generalised assignment command, reflecting the presence of non-rigid terms.

Definition 1. *The sets* $Fml_{DL}(\Sigma)$ *of* DL-formulas *and* $Prog_{DL}(\Sigma)$ *of* programs *are simultaneously defined as the minimal sets satisfying the following conditions:*

- $Atom(\Sigma) \subseteq Fml_{DL}(\Sigma)$.
- *If* ϕ_1, ϕ_2 *are in* $Fml_{DL}(\Sigma)$*, then so are* $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \to \phi_2$, $\phi_1 \leftrightarrow \phi_2$, *and* $\forall x\, \phi_1$ *and* $\exists x\, \phi_1$ *for all* $x \in Var$.
- *If* ϕ *is in* $Fml_{DL}(\Sigma)$ *and* p *is in* $Prog_{DL}(\Sigma)$*, then* $\langle p \rangle \phi$ *and* $[p]\phi$ *are in* $Fml_{DL}(\Sigma)$ *(in that case,* ϕ *is called the* scope *of* $\langle p \rangle$ *resp.* $[p]$*).*
- *If* t *is a non-rigid term and* s *is a term, then* $t := s$ *is in* $Prog_{DL}(\Sigma)$.
- *If* p_1, p_2 *are in* $Prog_{DL}(\Sigma)$*, then so is their sequential composition* $p_1; p_2$.
- *If* ψ *is a quantifier-free first-order formula and* p, q *are in* $Prog_{DL}(\Sigma)$*, then* $\mathbf{if}\, \psi\, \mathbf{then}\, p\, \mathbf{else}\, q\, \mathbf{fi}$ *and* $\mathbf{while}\, \psi\, \mathbf{do}\, p\, \mathbf{od}$ *are in* $Prog_{DL}(\Sigma)$.

Note, that all terms can occur within programs. In particular, we do not make any distinction between program variables and object variables.

In the following, we often do not differentiate between the modalities $\langle p \rangle$ and $[p]$, and we use $\{\!|p|\!\}$ to denote that a modality may be of either form.

The Kripke structures used to evaluate formulas from $Fml_{DL}(\Sigma)$ and programs from $Prog_{DL}(\Sigma)$ are called *DL-Kripke structures*. The set of states of a DL-Kripke structure $\mathcal{K}$ is obtained as follows: Let $\mathcal{A}_0$ be a fixed first-order structure for the rigid signature Σ_r, and let A denote the universe of $\mathcal{A}_0$. An n-ary function symbol $f \in \Sigma_r$ is interpreted as a function $f^{\mathcal{A}_0} : A^n \to A$ and every n-ary relation symbol $r \in \Sigma_r$ is interpreted as a set $r^{\mathcal{A}_0} \subseteq A^n$ of n-tuples. A *variable assignment* is a function $u : Var \to A$. We use $u[x/b]$ (where $b \in A$ and $x \in Var$) to denote the variable assignment such that $u[x/b](y) = b$ if $x = y$ and $u[x/b](y) = u(y)$ otherwise; moreover, if V is a set of variables, then $u_{|V}$ denotes the *restriction* of u to V. The set S of all *states* of $\mathcal{K}$ consists of all pairs $(\mathcal{A}, u)$, where u is a variable assignment and $\mathcal{A}$ is a first-order structure for the signature Σ, whose reduction to Σ_r, denoted by $\mathcal{A}_{|\Sigma_r}$, coincides with $\mathcal{A}_0$. We are now ready to define for each program p its interpretation $\rho(p)$, which is a relation on S (accessibility relation). Simultaneously, we define when a formula ϕ is true in a state $(\mathcal{A}, u)$, denoted by $(\mathcal{A}, u) \models \phi$.

Definition 2. *The* interpretation $\rho(p)$ *of programs* p *and the* relation $\models$ *between* S *and* $Fml_{DL}(\Sigma)$ *are simultaneously defined as follows:*

1. $(\mathcal{A}, u) \models \phi$ *is defined as usual in classical logic if* ϕ *is an atomic formula or its principal logical operator is one of the classical operators* $\wedge$, $\vee$, $\to$, $\leftrightarrow$, $\neg$, *or one of the quantifiers* $\forall$, $\exists$. *Also, the evaluation* $t^{(\mathcal{A},u)}$ *of terms* t *is defined as usual.*
2. $(\mathcal{A}, u) \models \langle p \rangle \phi$ *iff there is a pair* $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$ *with* $(\mathcal{B}, w) \models \phi$.

3. *$(\mathcal{A}, u) \models [p]\phi$ iff $(\mathcal{B}, w) \models \phi$ for all pairs $((\mathcal{A}, u), (\mathcal{B}, w))$ of states in $\rho(p)$.*
4. *If x is a variable, then $\rho(x := s) = \{((\mathcal{A}, u), (\mathcal{A}, u[x/s^{(\mathcal{A},u)}])) \mid (\mathcal{A}, u) \in S\}$.*
5. *If $t = f(t_1, \ldots, t_n)$ is a non-rigid term, then $\rho(t := s)$ consists of all pairs $((\mathcal{A}, u), (\mathcal{B}, u))$ such that $\mathcal{B}$ coincides with $\mathcal{A}$ except for the interpretation of f, which is given by*

$$f^{\mathcal{B}}(b_1, \ldots, b_n) = \begin{cases} s^{(\mathcal{A},u)} & \text{if } (b_1, \ldots, b_n) = (t_1^{(\mathcal{A},u)}, \ldots, t_n^{(\mathcal{A},u)}) \\ f^{\mathcal{A}}(b_1, \ldots, b_n) & \text{otherwise} \end{cases}$$

6. *$\rho(p_1; p_2)$ consists of all pairs $((\mathcal{A}, u), (\mathcal{C}, w))$ such that $((\mathcal{A}, u), (\mathcal{B}, v)) \in \rho(p_1)$ and $((\mathcal{B}, v), (\mathcal{C}, w)) \in \rho(p_2)$.*
7. *$\rho(\textbf{while}\, \psi\, \textbf{do}\, p\, \textbf{od})$ and $\rho(\textbf{if}\, \psi\, \textbf{then}\, p\, \textbf{else}\, q\, \textbf{fi})$ are defined as usual, e.g. [9].*

Definition 3. *A DL-Kripke structure $\mathcal{K}$ is a* model *of ϕ, denoted by $\mathcal{K} \models \phi$, iff $(\mathcal{A}, u) \models \phi$ for all states $(\mathcal{A}, u)$ of $\mathcal{K}$. A formula ϕ is* universally valid*, denoted by $\models \phi$, iff every DL-Kripke structure is a model of ϕ. A formula ϕ is* satisfiable *iff there are a structure $\mathcal{K}$ and a state $(\mathcal{A}, u)$ of $\mathcal{K}$ such that $(\mathcal{A}, u) \models \phi$.*

The particular choice of programs in $Prog_{DL}(\Sigma)$ (Def. 1) is rather arbitrary. The results being proved in this paper hold true for all choices of $Prog_{DL}(\Sigma)$ — including non-deterministic programming languages —, as long as Lemma 1 is guaranteed to hold. This lemma states that the effect of a program p is restricted to and only depends on the symbols occurring in p.

Lemma 1. *Let $\mathcal{K} = (S, \rho)$ be a DL-Kripke structure over a signature Σ, let p be a program, let V^p be the set of all variables occurring in p, and let Σ^p_{nr} be the set of all non-rigid function symbols occurring in p.*

1. *The program p only changes variables in V^p; i.e., if $x \notin V^p$ then $u(x) = w(x)$ for all $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$.*
2. *The program p only changes non-rigid symbols in Σ^p_{nr}; that is, if $f \notin \Sigma^p_{nr}$ then $f^{\mathcal{A}} = f^{\mathcal{B}}$ for all $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$.*
3. *The relation $\rho(p)$ is closed under changing variables not in V^p in the sense that, if $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$ and $u'_{|V^p} = u_{|V^p}$, then $((\mathcal{A}, u'), (\mathcal{B}, w')) \in \rho(p)$ for all w' with $w'_{|V^p} = w_{|V^p}$ and $w'_{|(Var\setminus V^p)} = u'_{|(Var\setminus V^p)}$.*
4. *The relation $\rho(p)$ is closed under changing the interpretation of non-rigid symbols not in Σ^p_{nr}, i.e., if $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$ and $\mathcal{A}'_{|\Sigma^p_{nr}} = \mathcal{A}_{|\Sigma^p_{nr}}$, then for all $\mathcal{B}'$ with $\mathcal{B}'_{|\Sigma^p_{nr}} = \mathcal{B}_{|\Sigma^p_{nr}}$ and $\mathcal{B}'_{|(\Sigma_{nr}\setminus\Sigma^p_{nr})} = \mathcal{A}'_{|(\Sigma_{nr}\setminus\Sigma^p_{nr})}$ we also have $((\mathcal{A}', u), (\mathcal{B}', w)) \in \rho(p)$.*

3 Extended Dynamic Logic with @*pre*

The Dynamic Logic defined in the previous section is suitable for the translation of OCL expressions that do not contain OCL's @*pre* operator. The detailed description of such a translation is outside the scope of this paper. It is important to note, however, that OCL properties (to which the @*pre* operator may be applied) are mapped into non-rigid function symbols in DL. That gives rise to the question of how a DL translation can be defined for OCL expressions containing the @*pre* operator.

The solution is to extend Dynamic Logic by adding to signatures for each non-rigid symbol f a new non-rigid symbol $f^{@pre}$. The new symbols have a special semantics that captures exactly the meaning of @*pre* in OCL: the interpretation of $f^{@pre}$ in the final state of a program execution is the same as that of f in the initial state of the program execution. Because of their special purpose, function symbols with @*pre* are not allowed to occur within programs.

Definition 4. *Let $\Sigma = \Sigma_r \cup \Sigma_{nr}$ be a signature. Then, the set of non-rigid function symbols of the extended signature $\Sigma^@ = \Sigma_r \cup \Sigma^@_{nr}$ is $\Sigma^@_{nr} = \Sigma_{nr} \cup \Sigma_{pre}$ where $\Sigma_{pre} = \{f^{@pre} \mid f \in \Sigma_{nr}\}$.*

The set $Fml_{DL^@}(\Sigma)$ of extended DL formulas *is defined as the set of DL formulas (Def. 1) over the extended signature $\Sigma^@$, except that programs must not contain any of the new symbols from Σ_{pre}, i.e., $Prog_{DL^@}(\Sigma) = Prog_{DL}(\Sigma)$.*

The semantics of formulas in $Fml_{DL^@}(\Sigma)$ is defined using DL-Kripke structures over the extended signature $\Sigma^@$.

Definition 5. *Let $\Sigma = \Sigma_r \cup \Sigma_{nr}$ be a signature, and let $\mathcal{K} = (S, \rho)$ be a DL-Kripke structure over Σ that is built using some first-order structure $\mathcal{A}_0$ over Σ_r.*

Then, the DL$^@$-Kripke structure $\mathcal{K}^@ = (S^@, \rho^@)$ is defined as follows: $S^@$ consists of all states $(\mathcal{A}, u)$ such that $\mathcal{A}$ is a structure for the extended signature $\Sigma^@ = \Sigma_r \cup \Sigma_{nr} \cup \Sigma_{pre}$ with $\mathcal{A}_{|\Sigma_r} = \mathcal{A}_0$, and u is a variable assignment. The accessibility relation $\rho^@$ is obtained from the accessibility relation ρ of $\mathcal{K}$ for all programs p by: $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho^@(p)$ iff

1. *$((\mathcal{A}_{|(\Sigma^@ \setminus \Sigma_{pre})}, u), (\mathcal{B}_{|(\Sigma^@ \setminus \Sigma_{pre})}, w)) \in \rho(p)$ and*
2. *$(f^{@pre})^{\mathcal{B}} = f^{\mathcal{A}}$ for all $f^{@pre} \in \Sigma_{pre}$.*

The relation $\models$ between states of $\mathcal{K}^@$ and formulas in $Fml_{DL^@}(\Sigma)$ is defined in the same way as the relation $\models$ between states of $\mathcal{K}$ and formulas in $Fml_{DL}(\Sigma)$ (Def. 2, 1.–3.), except that $\rho^@$ is used instead of ρ.

It usually only makes sense to use function symbols $f^{@pre} \in \Sigma^@_{nr}$ within the scope of some modality $\{\!|p|\!\}$, i.e., "after" the execution of some program p, because the purpose of $f^{@pre}$ is to refer to the value of f "before" executing p. Nevertheless, by definition, a symbol $f^{@pre} \in \Sigma^@_{nr}$ may as well be used outside the scope of any modality, in which case its semantics is similar to that of a rigid function symbol.

4 Transformations for Removing @*pre* from DL Formulas

The fixed interpretation of functions symbols in Σ_{pre} when occurring in the scope of a modality requires a special treatment by calculi for the extended DL. For example, the rule based on the equivalence of $\langle p \rangle \langle q \rangle \phi$ and $\langle p; q \rangle \phi$, which is part of most calculi for DL, cannot be used anymore. It is not sound in case ϕ contains a function symbol $f^{@pre} \in \Sigma_{pre}$ and the interpretation of the corresponding function symbol f is affected by the program p.

For practical reasons, however, we need a deductive calculus for the extended DL. Our solution to this problem, which allows to check satisfiability resp. validity of extended DL formulas, is to implicitly define a calculus for extended DL by providing a transformation from the extended into the non-extended version of DL (as defined in Section 2). That is, given a formula with occurrences of @*pre*, it is transformed into a formula without @*pre*, to which then a standard DL calculus can be applied.

In the following, we define two transformations on formulas of extended DL that allow to remove function symbols with @*pre* that occur in the scope of a modality (occurrences of symbols with @*pre* that are not within the scope of a modality are harmless and do not have to be removed). The first transformation (Sect. 4.2) is simpler but has also only weaker logical properties than the second transformation (Sect. 4.3).

4.1 Generalised Substitutions

The two transformations we define in the following are based on smaller local transformations, which can be seen as a generalised substitution $\psi[\![Orig/Subst]\!]$, where both $Orig$ and $Subst$ can be terms and function symbols.

Definition 6. *Let $\phi \in Fml_{DL^@}(\Sigma)$ be an extended DL formula over a signature $\Sigma = \Sigma_r \cup \Sigma_{nr}$, let $t, t' \in Term(\Sigma^@)$ be terms over the extended signature $\Sigma^@ = \Sigma_r \cup \Sigma_{nr} \cup \Sigma_{pre}$, and let $f, f' \in \Sigma^@$ be function symbols.*

The generalised substitution (of terms) *$\phi[\![t/t']\!]$ is defined as the result of replacing all those occurrences of t in ϕ by t' that (a) are neither within the scope of a modality $\{\!|p|\!\}$ nor within a program p and (b) do not contain any variables bound by a quantifier within ϕ.*

The generalised substitution (of function symbols) *$\phi[\![f/f']\!]$ is defined as the result of replacing all those occurrences of f in ϕ by f' that are neither within the scope of a modality $\{\!|p|\!\}$ nor within a program p.*

Example 1. The result of applying $[\![c/c_1]\!]$ to the formula $r(c) \wedge \langle c := c + 1\rangle r(c)$ is $r(c_1) \wedge \langle c := c + 1\rangle r(c)$. Applying $[\![s(x)/3]\!]$ to $\exists x\, s(x) \doteq 5$ yields $\exists x\, s(x) \doteq 5$.

4.2 Transformation Introducing New Function Symbols

The idea of the transformation τ_f (the subscript f indicates that τ_f introduces new function symbols) is to replace a function symbol $f^{@pre}$ by a new *rigid* function symbol f'.

Below, the transformation τ_f is formally defined only on a fragment of extended DL formulas, namely formulas of the form $\{\!|p|\!\}\phi$. Moreover, τ_f can only replace those occurrences of $f^{@pre}$ that are not within the scope of a nested modality in $\{\!|p|\!\}\phi$. We discuss the extension of τ_f to full extended DL at the end of this sub-section.

Definition 7. *Let $\Sigma = \Sigma_r \cup \Sigma_{nr}$ be a signature, let p be a program over Σ, let $\phi \in Fml_{DL^@}(\Sigma)$ be an extended DL formula, and let f' be a new rigid function symbol not occurring in Σ_r. We define*

$$\tau_f(\{p\}\phi) \quad \textit{as} \quad \forall x_1 \ldots \forall x_n\, f'(x_1, \ldots, x_n) \doteq f(x_1, \ldots, x_n) \wedge \{p\}\phi[\![f^{@pre}/f']\!]$$

where $x_1, \ldots, x_n$ are variables not occurring in $\{p\}\phi$.

The result of applying the transformation τ_f is an extended DL formula in $Fml_{DL^@}(\Sigma'_r \cup \Sigma_{nr})$ where $\Sigma'_r = \Sigma_r \cup \{f'\}$.

Theorem 1. *Using the notation from Definition 7, the following holds:*

1. *$\models \tau_f(\{p\}\phi) \to \{p\}\phi$ (i.e., $\tau_f(\{p\}\phi)$ is logically stronger than $\{p\}\phi$).*
2. *$\not\models \neg\{p\}\phi$ iff $\not\models \neg\tau_f(\{p\}\phi)$ (i.e., $\{p\}\phi$ is satisfiable iff $\tau_f(\{p\}\phi)$ is satisfiable).*

The logical relationship between the original and the transformed formula is exactly the same as, for example, between a first-order formula and its skolemised version. In both cases we extend the signature and get a logically stronger formula which is satisfiable iff the original formula is satisfiable.

The transformation τ_f (as defined above) can be used as the basis for a transformation that is applicable to all extended DL formulas. The idea of this more general transformation is to first convert the original formula ψ into an equivalent formula in negation normal form (NNF). The NNF property ensures that replacing in ψ_{NNF} a sub-formula ϕ by a formula ϕ' that (a) is logically stronger than and (b) satisfiability equivalent to ϕ yields a formula ψ'_{NNF} that has these two properties w.r.t. ψ_{NNF} and, thus, w.r.t. ψ (see [3] for details).

4.3 Transformation Introducing New Variables

The second transformation τ_v (the subscript v indicates that new variables are introduced) replaces terms of the form $f^{@pre}(t_1, \ldots, t_n)$ by new variables. Again, the transformation is formally defined only on a fragment of $Fml_{DL^@}(\Sigma)$. Now, we have the additional restriction that the terms $t_1, \ldots, t_n$ must not contain variables that are quantified in the formula to be transformed. Again, we discuss the extension of τ_v to full extended DL at the end of this sub-section.

Definition 8. *Let $\Sigma = \Sigma_r \cup \Sigma_{nr}$ be a signature, let $\phi \in Fml_{DL^@}(\Sigma)$ be an extended DL formula, let $f^{@pre}$ be a function symbol, let p be a program over Σ, and let $t_1, \ldots, t_n \in Term(\Sigma^@)$ be terms such that no t_i contains a variable quantified in ϕ. Now, let $y, x_1, \ldots, x_n$ be variables not occurring in $\{p\}\phi$. We define*

$$\begin{aligned}
\tau_v(\langle p\rangle\phi) \quad &\textit{as} \quad \exists y \exists x_1 \ldots \exists x_n\, y \doteq f(x_1, \ldots, x_n) \wedge \\
&\qquad \langle p\rangle \textstyle\bigwedge_{i=1}^{n}(x_i \doteq t_i) \wedge \phi[\![f^{@pre}(t_1, \ldots, t_n)/y]\!] \\
\tau_v([p]\phi) \quad &\textit{as} \quad \forall y \forall x_1 \ldots \forall x_n\, y \doteq f(x_1, \ldots, x_n) \to \\
&\qquad [p] \textstyle\bigwedge_{i=1}^{n}(x_i \doteq t_i) \to \phi[\![f^{@pre}(t_1, \ldots, t_n)/y]\!]
\end{aligned}$$

Theorem 2. *Using the notation from Definition 8, the following holds:*

$$\models \{p\}\phi \leftrightarrow \tau_v(\{p\}\phi).$$

The relationship between the original and the result of the transformation is stronger for τ_v than for the transformation τ_f presented in the previous subsection. It does not change the signature and, indeed, leads to a logically equivalent formula (which is all one can wish for). Consequently, τ_v can be recursively applied to sub-formulas of an extended DL formula.

Unfortunately, the main restriction on the applicability of τ_v, namely that the terms $t_1, \ldots, t_n$ must not contain quantified variables, cannot be lifted easily without destroying soundness of the transformation. Thus, for example, τ_v cannot be used to remove $f^{@pre}$ from the formula $\langle p \rangle \forall x\, r(f^{@pre}(x))$.

The problem of quantified variables can be tackled by shifting the quantifications of variables in the post-state formula to the outside of the modality $\{\!|p|\!\}$ (e.g., $\langle p \rangle \forall x \phi(x)$ becomes $\forall x \langle p \rangle \phi(x)$). In some cases, however, this technique requires the program p to be deterministic. To be more precise, when $\forall$ is shifted over $\langle p \rangle$ or $\exists$ is shifted over $[p]$, the result is in general not an equivalent formula for non-deterministic programs p (see [3] for details). In the above example, $\langle p \rangle \forall x\, r(f^{@pre}(x))$ is transformed into $\forall x \langle p \rangle r(f^{@pre}(x))$ (shift of quantification) and then into $\forall x \exists y \exists x_1\, y \doteq f(x_1) \wedge \langle p \rangle\, x_1 \doteq x \wedge r(y)$ (application of τ_v). Assuming p is indeed deterministic, the latter formula is equivalent to the original.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
3. T. Baar, B. Beckert, and P. H. Schmitt. An extension of Dynamic Logic for modelling OCL's @*pre* operator. TR 7/2001, University of Karlsruhe, Department of Computer Science, 2001.
4. B. Beckert. A Dynamic Logic for the formal verification of Java Card programs. In *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
5. D. Distefano, J.-P. Katoen, and A. Rensink. Towards model checking OCL. In *Proceedings, ECOOP Workshop on Defining a Precise Semantics for UML*, 2000.
6. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings, Asia Pacific Conference in Software Engineering*. IEEE Press, 1998.
7. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*. Reidel, 1984.
8. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
9. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
10. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.
11. Rational Software Corp. et al. *Unified Modelling Language Semantics, version 1.3*, June 1999. Available at: `www.rational.com/uml/index.jtmpl`.

Optimal Algorithms of Event-Driven Re-evaluation of Boolean Functions

Valeriy Vyatkin

Martin Luther University of Halle-Wittenberg,
Dept. of Engineering Science,
D-06099 Halle, Germany
phone: +49-(345)-552-5972; fax:+49-(345)-552-7304
Valeriy.Vyatkin@iw.uni-halle.de

Abstract. In many real-time applications, such as distributed logic control systems, response time is crucial. The response is generated by computation of Boolean functions. In this paper event-driven method of recomputations is suggested to get rid of computation overheads and provide the response in optimal time. New type of decision diagrams called Index Decision Diagrams (IDD for short) is introduced to facilitate such computations. Using IDD the computation of the function is performed in time, linear to the number of non-zero elements in the argument vector. Event-driven recomputation consists of two parts: *online recomputation* which is proven to have running time linear to the number of changed arguments, and *precomputation* which prepares the model for the former part in a fixed state of the arguments.

1 Introduction

This paper deals with the computational issues arisen in a class of real-time systems. A big deal of such systems as discrete controllers in industrial automation, have always been relying on massive Boolean computations. In these systems a controller communicates with the controlled system (called *plant*) by means of signals, which relay values of sensors to inputs of the controller and values of the outputs to the actuators of the plant. The controller is a computing device, implementing a control algorithm.

The control algorithm is usually represented in one of the programming languages specific for this field, implementation of which is reduced to the real-time computation of a (huge) number of Boolean functions.

The usual way of the computation is cyclic. First, the current status of the plant, which is indicated by sensors, is stored in an input buffer, then the whole control program (Boolean functions) is executed, while the values of inputs in the buffer remain unchanged, and in the last step the calculated outputs are transmitted to the actuators of the plant. Such a procedure, called *scan* repeats itself over and over again. The duration of the scan determines the response characteristic of controller. The shorter response is, the better the quality and reliability of the control are expected.

The latest trends in the development of control systems urgently require changes in these "cyclic" computations. We mention here two main reasons:

1. The control systems become distributed. As a consequence the data are delivered to/from controllers not directly, but via a network as events, i.e. messages about changes of the inputs/outputs. The new being developed standard for distributed controller design IEC61499, introduced in [4], largerly relies upon the event-driven implementation. This requires updated methods of computations.
2. The control algorithms themselves are getting more complicated. The supervisory control theory, introduced by Ramadge and Wonham [5], suggests to divide the controller onto two parts: the sequential controller which reflects the required processing cycle, and the supervisor, which observes the current situation and prevents the control system from getting to dangerous states. It is possible to build such supervisors automatically, given a formal model of a controlled plant and a formal description of the notion of "danger". The resulting supervisors, however, turn to be huge arrays of very complicated Boolean functions. Computation of supervisors is even more complicated, when the latter is placed to the distributed environment mentioned above.

In this paper we attempt to suggest a way of computation corresponding to the new challenges.

2 Re-evaluation versus Evaluation

Binary Decision Diagram (BDD) is a directed acyclic graph (dag) with a single root, introduced in [6] for Boolean function representation. Evaluation of a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is performed by a branching program with the structure of the BDD given an instance of input argument $X \in \{0,1\}^n$ in time $O(n)$ (for restricted BDD). This way of computation is also called start-over or full evaluation.

In this paper we introduce the BDD derivative termed Index Decision Diagram (IDD) which computes $f(X)$ in time linear to the "number of ones" in the vector X. Instead of dealing with the input vector represented as a Boolean vector, it is more compact to use its compressed form of the ordered list of indexes $\lambda(X)$ of the elements equal to 1. For example, $\lambda(\langle 0000100001 \rangle) = \langle 5, 10 \rangle$. The IDD is a BDD modification intended to process the input represented this way.

The IDD application can be beneficial for the computation if the input array is sparse, i.e. one value, for example zeros, predominate over the other (ones) in every input instance. The other application which we are going to show in this paper, is the event-driven re-evaluation of a Boolean function. As opposed to the evaluation, the re-evaluation finds $f(X_{new})$ given the change $\Delta \in \{0,1\}^n$ to the input X_{old} such that $X_{new} = X_{old} \oplus \Delta$. For example, $X_{old} = \langle 00010001 \rangle$, $\Delta = \langle 01000001 \rangle$, and $X_{new} = \langle 01010000 \rangle$. In many applications the change occurs just in a few bits of the input array at once, so Δ is very sparse Boolean array and the IDD application seems to be reasonable.

When the re-evaluation is applied to the event-driven systems it is assumed that the change Δ is induced by an event. We suggest to use some *precomputation*, placed between events to prepare some auxiliary data which is used upon events to accelerate the *on-line re-evaluation*. It is important to minimize both parts, with the emphasis on the on-line part which is especially critical for the response characteristic of many discrete-event applications, such as logic control, etc. Usually in such applications events occur relatively seldom, so there is enough time for the pre-computations. However, once an event occurred the reaction must be as fast as possible. Certainly, re-evaluation is worthwhile when compared to the evaluation if it can be performed in substantially less time than $O(n)$.

In this *event-driven* framework, the start-over algorithm can be regarded as having zero time precomputation and on-line re-evaluation of $O(n)$ time. Another example of the event-driven algorithm, introduced in [7], precomputes $f(X_{old} + \Delta)$ for the subset $\{\Delta : |\lambda(\Delta)| \leq d\}$ and stores the precomputed values in the d-dimensional table. The precomputation takes $O(n^{d+1})$ time since the table has $O(n^d)$ entries and $O(n)$ time is required for each entry to be computed. Upon event, the value corresponding to the particular $\delta = \lambda(\Delta)$ can be restored from the table in time linear to the length of the list $O(|\delta|)(|\delta| \leq d)$.

The on-line algorithm presented in this paper uses the Index Decision Diagram to compute the result in time $O(|\delta|) = O(|\lambda(\Delta)|)$. Precomputation is used to compose the IDD given a BDD and values of arguments X. Upon event, the algorithm finds value $f(X_{old} \oplus \Delta)$ using the IDD and given δ. The problem of function's re-evaluation is related also to the *incremental* methods of computations. More specifically, re-evaluation using BDD is a particular case of the incremental circuit annotation problem [1,3].

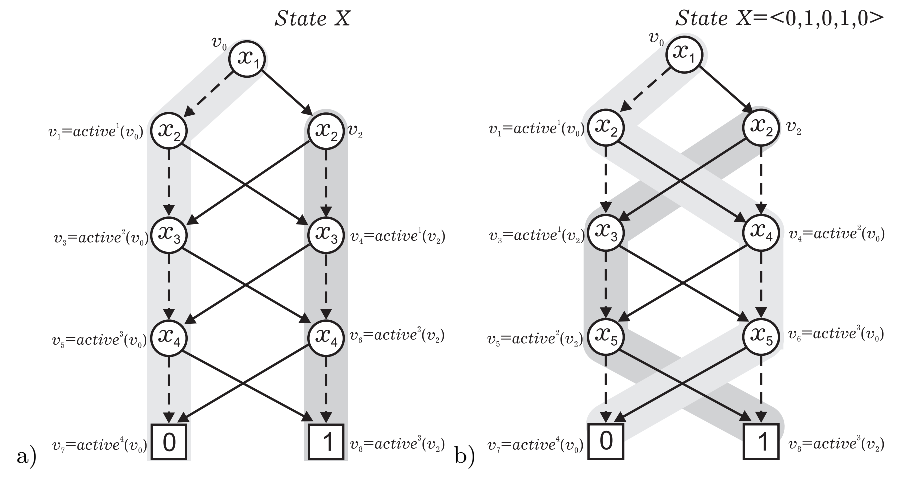
3 Computation of Boolean Functions Using BDD

Let v_0 and V denote respectively the root and the set of vertices of a BDD. Each non-terminal vertex $v \in V$ has exactly two outgoing edges called $0-$ and $1-$ edge. When drawing a BDD, the 0-edge is depicted as a dotted line, and the 1-edge as a solid line. Target of the 0-edge is termed $lo(v)$ and of the 1-edge $hi(v)$. A non-terminal vertex is also associated with an input variable $x_{v.ind} \in X$ specified by the index $v.ind(1 \leq v.ind \leq n)$.

A BDD has exactly two terminal vertices, associated with constant $v.value \in \{0,1\}$ and assigned the pseudo index $v.ind \equiv n+1$. Each vertex v of the BDD denotes a Boolean function f_v as follows: in the terminal vertices $f_v \equiv v.value$, in the non-terminal ones it is defined in a recurrent way:

$$f_v = \overline{x_{v.ind}} f_{lo(v)} \vee x_{v.ind} f_{hi(v)}. \quad (1)$$

The function f_{v_0} denoted in the root is said to be denoted by the whole BDD. There is a two-way relationship between functions and diagrams - each Boolean function also can be expanded into a BDD by iterative application of the Shannon expansion [2].



State X
v_0 x_1
v_1=active$^1(v_0)$ x_2
x_2 v_2
v_3=active$^2(v_0)$ x_3
x_3 v_4=active$^1(v_2)$
v_5=active$^3(v_0)$ x_4
x_4 v_6=active$^2(v_2)$
a) v_7=active$^4(v_0)$ 0
1 v_8=active$^3(v_2)$
State X=<0,1,0,1,0>
v_0 x_1
v_1=active$^1(v_0)$ x_2
x_2 v_2
v_3=active$^1(v_2)$ x_3
x_4 v_4=active$^2(v_0)$
v_5=active$^2(v_2)$ x_5
x_5 v_6=active$^3(v_0)$
b) v_7=active$^4(v_0)$ 0
1 v_8=active$^3(v_2)$

Fig. 1. Binary decision diagram with outlined active paths in the state $X = \langle 0, 0, 0, 0\rangle$ (a) and after change $\delta = \langle 2, 4\rangle$, i.e. at $X + \delta = \langle 0, 1, 0, 1\rangle$ (b)

A BDD is *restricted* (RBDD for short) if no variable is associated with more than one vertex in any directed path. Therefore length of any directed path in a RBDD is bounded by the size of the input array $|X| = n$. A BDD is *ordered* (OBDD) if in any directed path the indexes of the associated variables follow to the increasing order. Obviously, an ordered BDD is also restricted.

At a fixed input $X \in \{0, 1\}^n$ one of the edges $(v, lo(v)), (v, hi(v))$ is called *active*. If $x_{v.ind} = 0$ then $(v, lo(v))$ is active, otherwise the opposite edge $(v.hi(v))$ is active. The destination of the active edges is called the active successor of the vertex: $w = active(v)$.

Let $active^i(v) = \begin{cases} active(v) & \text{if} \quad i = 1 \\ active^{i-1}(v) & \text{if} \quad i > 1 \end{cases}$

denote the $i - th$ active successor of v. A directed path starting in v, formed only of active edges, and ending in a terminal vertex is denoted

$$v.\mathbf{P}_X = \left\langle v, active(v), active^2(v), \ldots , active^p(v)\right\rangle$$

and called the active path of vertex v at input X. The subscript X emphasizes the dependence of the active path on the value of the current input. In particular if the input array contains only zeros, i.e. $X = \mathbf{0}$ the active path is called zero-path of the BDD. A vertex v is called *source of active path* if it has no active incoming edges. An active path which starts in a source vertex is called *full active path*.

Figure 1-a presents an example of OBDD for the function

$$f = \overline{(x_1 \oplus x_2)}(x_3 \oplus x_5) \vee (x_1 \oplus x_2)\overline{(x_4 \oplus x_5)}.$$

Active edges corresponding to $X = \mathbf{0}$ are gray shadowed. The full active paths in this state of X are rooted in v_0 and v_2: $v_0.\mathbf{P} = \langle v_0, v_1, v_3, v_5, v_7 \rangle$, $v_2.\mathbf{P} = \langle v_2, v_4, v_6, v_8 \rangle$ respectively.

The expression (1) can be transformed in terms of the active successor as follows:

$$f_v(X) = f_{active(v)(X)}. \tag{2}$$

The recursive computation of the function according to (2) can be performed by traverse of the BDD. The traverse starts in the root $v = v_0$ and always chooses the active child of the current vertex $active(v)$ to be continued. The value of the constant associated with the terminal vertex of the active full path $v.\mathbf{P}_X$ determines the current value of the function $f_v(X)$. Therefore, time of the full computation of function using RBDD is bounded by $O(n)$.

4 Index Decision Diagrams

Let $\lambda = \lambda(X)$ denote an ordered list of indexes of ones in X. For example if $X = \langle 0000100101 \rangle$ then $\lambda(X) = \langle 5, 8, 10 \rangle$. In this section we introduce the Index Decision Diagram (IDD) of a Boolean function which enables to compute $f(X)$ in $O(|\lambda(X)|)$ time given the list $\lambda(X)$.

Let G be an ordered BDD which denotes the function f, and $v.\mathbf{P_0}$ the *zero path.* We define the *search mapping* $M : V \times \{1, 2, .., n\} \to V$ as follows: for given $v \in V$ and $\forall i \in v.ind \dots n+1$: $M_v(i)$ designates the vertex with minimum index greater than or equal to i in $v.\mathbf{P_0}$. The Index Decision Diagram (IDD) $\mathcal{E} = \mathcal{E}(G)$ is a graph which is built for a given BDD G as follows:

1. IDD and BDD have the same set of vertices $V_{\mathcal{E}} = V_G$.
2. A non-terminal vertex $v \in V_{\mathcal{E}}$ has $n - v.ind + 2$ outgoing edges defined by array of their targets $lihk_v[v.ind..n+1]$ such that: $link_v[v.ind] = hi(w)$, and the others $n - v.ind + 1$ edges are defined as $link[i] = M_v(i), i = v.ind + 1, .., n + 1$.

An example of IDD for the OBDD from Figure 1-a is presented in Figure 2. Note that since the zero-path with source v_0 does not contain vertex with variable x_4, the corresponding links in the vertices v_0, v_1, v_3 are redirected to the vertex associated with x_5. Similarly, in the zero-path with source in v_2 variable x_3 is not included and the link in v_2 is redirected to v_4.

The function denoted by the OBDD rooted in a vertex v depends on $x_{v.ind}, \dots, x_n$. Let us represent the input subarray $X[v.ind..n]$ as a concatenation of some "leading zero's" subarray $\mathbf{0}_{v.ind..i}$ and the remainder $X[i..n]$:

$$X[v.ind..n] = \mathbf{0}_{[v.ind..i-1]} \cdot X[i..n].$$

If X is represented in such a way, i.e. if $x_{v.ind} = x_{v.ind+1} = .. = x_{i-1} = 0$, the following proposition holds:

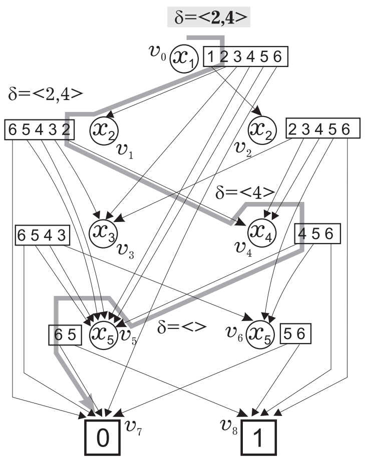


Fig. 2. Evaluation of IDD given the list $\lambda = \langle 2, 4\rangle$

Proposition 1. $f_v(X[v.ind..n]) = f_{M_v(i)}(X[i..n])$

Proof. Let us prove the statement by induction on i. If $i = v.ind$ then $M_v(i) = M_v(v.ind) = hi(v)$. According to (1) if $x_{v.ind} = 1$ then $f_v = f_{hi(v)}$ so the statement holds.

Assume that the statement holds for $i = k > v.ind$ and prove it for $i = k+1$. Denote $w = M_v(k)$. According to the definition of M_v, w is such that $w.ind$ is minimum $w.ind \geq k$.

If $w.ind = k$ then $lo(w).ind \geq k+1$ so $M_v(k+1) = lo(w)$. In case of $i = k+1$, $x_k = 0$ which implies $f_{M_v(k)} = f_w = f_{lo(w)} = f_{M_v(k+1)}(X[k+1..n])$.

If $w.ind > k$ (i.e $w.ind \geq k$) then $M_v(k+1) = M_v(k) = w$ and $f_{M_v(k+1)} = f_{M_v(k)}$.

Now suppose that the input vector is represented by the list $\lambda = \lambda(X)$. If root of the BDD (and IDD) is v then w.l.o.g. we can assume $X = X[v.ind..n]$ and $min(\lambda) \geq v.ind$. Then

$$X = \mathbf{0}_{[v.ind,min(\lambda)-1]} \cdot X[min(\lambda)..n]$$

and according to the proposition 1 $f_v = f_{M_v(min(\lambda))}(X[min(\lambda)..n])$.

The evaluation of $f(X)$ using IDD is performed by the algorithm presented in Figure 3. The input of the algorithm is list λ, the output is the value of the function. Main loop [2]-[6] iterates no more than $2|\lambda|$ times. In the body of the loop the current vertex v moves to $v' = M_v(\min(\lambda))$ in which $f_{v'} = f_v$. After that λ is adjusted to not contain any number less than $v'.ind$ so that the invariant $v.ind \geq min(\lambda)$ of the loop always holds. If the loop halts at $v : v.ind < n+1$ then there are no more indexes in λ which means that $x_{v.ind} = x_{v.ind+1} = .. = x_n = 0$. Therefore $f_v = f_v(\mathbf{0}_{v.ind..n}) = f_{M_v(n+1)}$. This is done in line [8].

```
**Algorithm** *IDD evaluation* ($\lambda$: list of integer)
**begin**
[1] $v$:=Root of the IDD;
[2] **while** $\lambda$ is not empty **do**
[3]         *Loop invariant is:* $\{v.ind \le min(\lambda)\}$
[4]         $v' := M_v(min(\lambda))$
[5]         Delete from $\lambda$ all numbers less than $v'.ind$
[6]         $v := v'$
[7] **end**{ **while** }
[8] **if** $v.ind < n + 1$ **then** $v := M_v(n + 1)$
[9] Result is $v.value$
**end**
```

Fig. 3. Algorithm re-evaluates function using as input the IDD and the list λ of indexes of variables equal to 1

Since the minimum of the ordered λ is $\lambda[1]$ and computation of $M_v(min(\lambda))$ is performed by the lookup in the linear array *link* in a constant time, the body of the loop takes constant time, so the total computation is done in time linear to $2|\lambda|$. A little modification of the mapping M_v is able to reduce the number of iterations to $|\lambda|$: assume $M_v^{'}(i) = M_v(i)$ for all i such that the corresponding x_i is not presented in the zero-path $v.\mathbf{P}_0$, and $M_v^{'}(i) = hi(M_v(i))$ if x_i is in the zero-path. Then each iteration of the loop [2-6] makes $min(\lambda) > v.ind$ which guarantees at least one index from λ to be deleted at each iteration thus bounding their number by $|\lambda|$. However we sacrifice this improvement to the clarity of explanation.

The algorithm of IDD evaluation is illustrated in Figure 2 for $X = \langle 0, 1, 0, 1, 0\rangle$, i.e. at $\lambda = \langle 2, 4\rangle$. In the begin $v = v_0$,$min(\lambda) = 2$ and v moves to $v' = M_v(2) = v_1$. In $v = v_1$ we have $M_v(2) = v_4$ so that 2 to be deleted from λ since $v_4.ind = 4 > min(\lambda) = 2$. Then $v := M_{v_4}(4) = v_5$ and 4 is also to be deleted from λ. In this step the list λ is exhausted, so according to the line [9] of the algorithm the result can be found as *value* attribute of vertex $M_{v_5}(6) = v_8$.

The following theorem summarizes all stated above about the IDD evaluation:

Theorem 1. *The value of Boolean function $f(X)$ can be computed by the algorithm of IDD evaluation in time linear to the number of ones in the argument vector X using its presentation as an IDD.*

5 Generation of IDD

To build the IDD for a given OBDD G it is required to fill in every vertex $v \in V_G$ the arrays $link_v$ of pointers defining the edges going out of v in IDD.

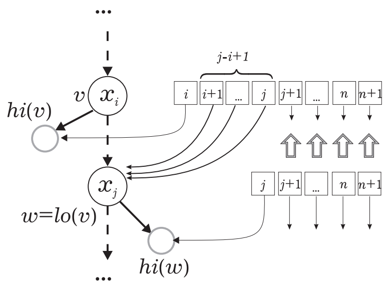


Fig. 4. IDD edges in vertex v are formed using those of its active successor w in time, linear to $n - i + 2$

Consider two adjacent vertices v and w in the OBDD, such that $w = lo(v)$, $v.ind = i$,$w.ind = j$ and $j > i$ as shown in Figure 4. According to the definition of IDD edges, in the vertex v:

$$\begin{array}{ll} v.link[i] & = hi(v), \\ v.link[i+1, \dots, j] & = w, \\ v.link[j+1 \dots n+1] & = w.link[j+1 \dots n+1] \end{array}$$

Therefore the part $v.link[j+1 \dots n+1]$ of the IDD links in v can be copied from those of w, while the others can be assigned each in a constant time. Complexity of the *link*s generation, if the above routine is applied in down-up order from the vertices with higher indexes to the vertices with lower indexes is linear in the sum of links' number by all vertices of the IDD (and OBDD):

$$|E_{\mathcal{E}}| = \sum_{v \in V_G} n - v.ind + 2 \tag{3}$$

In general the sum is upper bounded by $O(n\,|V_G|)$.

6 Re-evaluation of Boolean Function Using IDD

Let us denote $x^\alpha = \overline{x}$ if $\alpha = 1$ and $x^\alpha = x$ if $\alpha = 0$. Obviously, $0^\alpha = \alpha$ and $\alpha^\alpha = 0$. Let $X = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$. The *normalized* function f_X is derived from a Boolean function f as follows: $f_X(x_1, x_2, \dots, x_n) = f(x_1^{\alpha_1}, x_2^{\alpha_2}, \dots, x_n^{\alpha_n})$. Value of the normalized function $f_X(\mathbf{0})$ with all-zeros argument $\mathbf{0} = 0, 0, \dots, 0$ is equal to the $f(X)$: $f_X(\mathbf{0}) = f(0^{\alpha_1}, 0^{\alpha_2}, \dots, 0^{\alpha_n}) = f(\alpha_1, \alpha_2, \dots, \alpha_n)$.

The re-evaluation is required to compute the function after a change Δ is occured with the input: $X_{new} = X \oplus \Delta$. The event-driven re-evaluation consists of the on-line part which follows the event denoted as a list $\delta = \lambda(\Delta)$ and evaluates the new value of the function, and precomputation which is placed between events and provides the on-line part with required auxiliary data.

In our case the auxiliary data consists of the IDD for the normalized function f_X. It is clear that $f(X_{new}) = f_X(\Delta)$. As it follows from the previous section, having the IDD for the f_X the value of $f_X(\mathit{\Delta})$ can be computed in time $O(|\lambda(\mathit{\Delta})|)$. The OBDD for the normalized function can be derived from the OBDD denoting f trading places of its $0-$ and $1-$ edges in all the vertices $v : x_{v.ind} = 1$. It can be easily observed for a single-variable case with OBDD with one non-terminal vertex and then proved by induction for an arbitrary OBDD. Time of the transformation is bounded by $O(V)$. Given the OBDD for f_X, the IDD for f_X is composed in $O(n\,|V|)$ time, so the total precomputation required is bounded by $O(|V| + n\,|V|) = O(n\,|V|)$. Given the precomputed IDD and the list δ of indexes of changed variables the re-evaluation requires $O(|\delta|)$ time to find the new value of the function.

We summarize this result as the following theorem:

Theorem 2. *On-line re-evaluation of a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ after a change $\mathit{\Delta}$ to the input X can be done in time $O(|\lambda(\mathit{\Delta})|)$ at the precomputation of $O(|V|\,n)$ time.*

References

1. B. Alpern, R. Hoover, B.K. Rosen, P.F Sweeney, and K. Zadeck. *Incremental evaluation of computational circuits*, pages 32–42. Proceedings of First Annual ACM-SIAM symposium on Discrete Algorithms, 1990.
2. C.E.Shannon. A symbolic analysis of relay and switching circuits. *Trans. AIEE*, 57:713–723, 1938.
3. Ramalingam G. *Bouded Incremental Computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1996.
4. J.H.Christensen. Basic concepts of IEC 61499. In *Proc. of Conference "Verteile Automatisieriung" (Distributed Automation)*, pages 55–62, Magdeburg, Germany, 2000.
5. Ramadge P.J. and Wonham W.M. Supervisory control of a class of discrete event processes. *SIAM J Control Optimisation*, 25(1):206–230, 1987.
6. S.B.Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–16, 1978.
7. V.Viatkin, K.Nakano, and T.Hayashi. *Optimized Processing of Complex Events in Discrete Control Systems using Binary Decision Diagrams*, pages 445–450. Int'l workshop on algorithms and architectures in real-time control, IFAC, 1997.

A Transformation of SDL Specifications — A Step towards the Verification

Natalia Ioustinova[1] and Natalia Sidorova[2]

[1] Department of Computer Science
University of Rostock, Alb. Einstein Str. 21, D-18059 Rostock, Germany
fax +49(0)381 498 3440
ustin@informatik.uni-rostock.de

[2] Dept. of Math. and Computer Science, Eindhoven University of Technology,
PO Box 513, 5600 MB Eindhoven, The Netherlands
fax +31(0)40 246 3992
n.sidorova@tue.nl

Abstract. Industrial-size specifications/models (whose state space is often infinite) can not be model checked in a direct way — a verification model of a system is model checked instead. Program transformation is a way to build a finite-state verification model that can be submitted to a model checker. Abstraction is another technique that can be used for the same purpose. This paper presents a transformation of SDL timers aimed at the reduction of the infinite domain of timer values to a finite one with preserving the behaviour of a system. A timer abstraction is proposed to further reduce the state space. We discuss the ideas behind these transformations and argue their correctness.

1 Introduction

Model checking [3] is one of the most popular and successful verification techniques accepted both in academia and in industry. One of the main reasons of its success is its promise to automatically check a program against a logical specification, typically a formula of some temporal logic. A stumbling-block limiting the model-checking application area is the notorious state-space explosion. The major factors influencing the state space are, clearly, the size and the complexity of a specification. In many cases, abstractions and compositional techniques make it possible to cope with the state-space explosion and apply model checking to real-life industrial systems (see e.g., [12]). However, besides size and complexity, there exists another factor cumbering verification.

Development of a specification language, its semantical concept are greatly affected by the intended mode of its use, its application domain. Often, the final objective is to provide an executable specification/implementation. In that case, the specification language and its semantics should provide a framework for constructing faithful and detailed descriptions of systems. No wonder that specifications written in these implementation-oriented languages are harder to verify than the ones written in the languages developed as input languages for model checkers. In this paper, we concentrate on some aspects of modelling time in the implementation-oriented languages, taking SDL (Specification and Description Language) [11] as an instance of this class of languages.

SDL is a popular language for the specification of telecommunication software as well as aircraft and train control, medical and packaging systems. Timing aspects are very important for these kinds of systems. Therefore, behaviour of a system specified in SDL is scheduled with the help of timers involved into the specification. The model of SDL timers was induced by manners of implementation of timers in real systems. An SDL timer can be activated by setting it to a value ($\texttt{NOW} + \delta$) where expression `NOW` provides an access to the current system time and δ is a delay after which this timer expires, i.e., the timer expires when a system time (system clock) reaches the point ($\texttt{NOW} + \delta$). Such an implementation of timers immediately means that the state space of SDL-specifications is infinite just due to the fact that timer variables take an infinite number of growing, during the system run, values. An inverse timer model is normally employed in the verification-oriented languages: a timer indicates a delay left until its expiration, i.e., a timer is set to value δ instead of ($\texttt{NOW} + \delta$), and this value is decreased at every tick of the system clock. When the timer value reaches zero, the timer expires. This model of timers guarantees that every timer variable takes only a finite (and relatively small) number of values.

Another SDL peculiarity that adds to the complexity of verification is the manner the timers expire in SDL. SDL is based on the Communicating Extended State Machines; communication is organized via the message passing. For the uniformity of communication, timers are considered as a special kind of signals and a process learns about a timer expiration by dint of a signal with the name of the expired timer, inserted in the input port of the process. From the verification point of view it would be better if a timer expiration had been diagnosed by a simple check of the timer value.

Though formal verification of SDL-specifications is an area of rather active investigations [2,8,6,10,14], the time-concerned difficulties were being got round for a long time by means of abstracting out time and timers. Due to engineering rather than formal approaches to constructing abstractions, some of proposed abstractions turned out to be not safe (see [2] for details). In [2], a toolset and a methodology for the verification of *time-dependent* properties of SDL-specifications are described. The SDL-specifications are translated into DT Promela, the input language of the DT Spin (Discrete Time Spin) model checker [1], and then verified against LTL formulas. Some arguments in favour of a behavioural equivalence of the DT Promela translation to the original specification are given.

Here, we propose a transformation of SDL specification into SDL itself, where the new `timer` variable type is substituted for the traditional SDL `timer` type. The underlying idea is similar to the one in [2], but providing SDL to SDL transformation, we make the transformation principles independent of a particular model checker, and the formal proof of model equivalence substantiate that the transformed model can be safely used for the verification.

Admitting that in a number of cases timer abstractions are useful[1], we believe that the safe timer abstraction of [2] does not yield a desirable result in a number of situations because it abstracts not just timers but time. Here we propose a more flexible w.r.t. the refinement degree abstraction, for which the abstraction of [2] is a particular case.

The paper is organised as follows. In Section 2 we outline the SDL semantics we use. In Section 3 we present a behaviour-preserving transformation for SDL-specifications. In Section 4 we propose a timer abstraction. We conclude in Section 5 by evaluating the results.

2 SDL Time Semantics

SDL is a general purpose description language for communicating systems. The basis for the description of a system behaviour is Communicating Extended State Machines represented by processes. A process consists of a number of states and a number of transitions connecting the states. The input port of a process is an unbounded FIFO-queue. Communication is based on the signal exchange between the system and its environment and between processes within the system.

A specification *Spec* is the parallel composition $\Pi^n_{i=1} P_i$ of a finite number of processes. A process P is specified as a tuple $(Var, Timer, Loc, Edg, q, \gamma_0)$, where Var is a finite set of variables, $Timer$ is a finite set of timers, Loc is a finite set of control states, or *locations*, q is a process queue, and γ_0 is an initial configuration. A *valuation* of process variables maps a process variables to values, i.e., $Var \longrightarrow D$, where D is some data domain. A *configuration* γ of a process is a control state l together with a valuation of variables θ, a valuation of timers ϕ, and a process input queue q. We denote the set of valuations by Val and the set of configurations by Γ. The set of *edges* $Edg \subseteq Loc \times Act \times Loc$ describes changes of configuration resulting from performing an action from a set of actions Act.

We distinguish the following actions: (i) *input* of a signal s containing a value to be assigned to a local variable, (ii) *output* of a signal s together with a value described by an expression to a process P', (iii) *assignments*, (iv) *setting* and *resetting* a timer. In Loc, we distinguish a set Loc_i of input locations, i.e., the locations where input actions are allowed. Note that it is the only sort of actions allowed in these locations. We assume a boolean guard g to be imposed on each action, and in every location, besides input locations, at least one guard is evaluated to *true*. The input actions are denoted as $g \rhd ?s(x)$, outputs as $g \rhd P'!s(v)$, assignments as $g \rhd x :\ = exp$, setting and resetting of a timer t as $g \rhd \mathtt{SET}(exp, t)$ and $g \rhd \mathtt{RESET}(t)$ respectively.

The step semantics of a process is a labelled transition relation on configurations, $\longrightarrow_\lambda \subseteq \Gamma \times Label \times \Gamma$. Internal steps are denoted by τ, the time-progress steps by *tick* and communication steps are labelled by a triple of process, signal and value to be transmitted. The behaviour of a single process is given by

[1] If a property is expected to hold independently of the settings of a timer, for example.

Table 1. Step semantics for process P

$$\frac{l \longrightarrow_{g\,\triangleright ?s(x)} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad l \in Loc_i \qquad s \notin Sig_{timeout}}{(l, \theta, \phi, s(v) :: q) \to_\tau (\hat{l}, \theta_{[x \mapsto v]}, \phi, q)}\ \textsc{Input}$$

$$\frac{l \longrightarrow_{g\,\triangleright ?\mathtt{NONE}} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad l \in Loc_i}{(l, \theta, \phi, q) \to_\tau (\hat{l}, \theta, \phi, q)}\ \textsc{NoneInput}$$

$$\frac{l \longrightarrow_{g\,\triangleright ?s'(x)} \hat{l} \in Edg \Rightarrow s' \neq s \qquad l \in Loc_i \qquad s \notin Sig_{timeout}}{(l, \theta, \phi, s(\_) :: q) \to_\tau (l, \theta, \phi, q)}\ \textsc{Discard}$$

$$\frac{v \in Val}{(l, \theta, \phi, q) \longrightarrow_{P?s(v)} (l, \theta, \phi, q :: s(v))}\ \textsc{Receive}$$

$$\frac{l \longrightarrow_{g\,\triangleright P'!s(e)} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad [\![e]\!]_\theta = v \qquad l \notin Loc_i}{(l, \theta, \phi, q) \to_{P'!s(v)} (\hat{l}, \theta, \phi, q)}\ \textsc{Output}$$

$$\frac{l \longrightarrow_{g\,\triangleright\ x\,:=\,exp} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad [\![exp]\!]_\theta = v \qquad l \notin Loc_i}{(l, \theta, \phi, q) \to_\tau (\hat{l}, \theta_{[x \mapsto v]}, \phi, q)}\ \textsc{Assign}$$

$$\frac{l \longrightarrow_{g\,\triangleright\,\mathtt{SET}(exp,t)} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad [\![exp]\!]_\theta = v \qquad l \notin Loc_i}{(l, \theta, \phi, q) \to_\tau (\hat{l}, \theta, \phi_{[t \mapsto on(v)]}, \pi_t(q))}\ \textsc{Set}$$

$$\frac{l \longrightarrow_{g\,\triangleright\,\mathtt{RESET}(t)} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad l \notin Loc_i}{(l, \theta, \phi, q) \to_\tau (\hat{l}, \theta, \phi_{[t \mapsto off]}, \pi_t(q))}\ \textsc{Reset}$$

$$\frac{[\![\mathtt{NOW}]\!]_\gamma = v \qquad [\![t]\!]_\phi = on(v)}{(l, \theta, \phi, q) \to_\tau (l, \theta, \phi_{[t \mapsto off]}, q :: t)}\ \textsc{Expiration}$$

$$\frac{l \longrightarrow_{g\,\triangleright ?t} \hat{l} \in Edg \qquad [\![g]\!]_\theta = true \qquad l \in Loc_i \qquad t \in Sig_{timeout}}{(l, \theta, \phi, t :: q) \to_\tau (\hat{l}, \theta, \phi, q)}\ \textsc{TInput}$$

$$\frac{l \longrightarrow_{g\,\triangleright ?s(x)} \hat{l} \in Edg \Rightarrow s \neq t \qquad l \in Loc_i \qquad t \in Sig_{timeout}}{(l, \theta, \phi, t :: q) \to_\tau (l, \theta, \phi, q)}\ \textsc{TDiscard}$$

sequences of configurations $\gamma_0 \to_\lambda \gamma_1 \to_\lambda \ldots$. Table 1 gives the step semantics of a process P, where we assume $Sig_{timeout}$ to be the set of timer signals. We write ε for an empty queue, $s(v) :: q$ for a queue with message $s(v)$ at the head of the queue, i.e., $s(v)$ is the message to be read from the queue next; $q :: s(v)$ denotes a queue with $s(v)$ at the tail. The notation $\theta_{[x \mapsto v]}$ stands for the valuation equalling θ for all $y \in Var \backslash \{x\}$ and mapping a variable x to a value v.

The rule Discard captures a specific feature of SDL92: if the signal from the head of the queue does not match any input defined as possible for the current (input) location, the signal is removed from the queue without changing the location and the valuation. NoneInput defines the SDL `NONE` transition, which is a spontaneous transition from a location l to the location $\hat{l}$.

In SDL, the concept of timers is employed to specify timing conditions imposed on a system. Two data types, `Time` and `Duration`, are used to specify time values. A variable of the `Time` type indicates some point of time. A variable of the `Duration` type represents a time interval. A process can access the current system time by means of the `NOW` operator, which we replace for simplicity by shared variable `NOW`. Each timer is related to a process instance; a timer is either active (set to a value) or inactive (reset). Two operations are defined on timers: `SET` and `RESET` (rules Set and Reset). A timer is activated by setting it to a value ($\mathtt{NOW} + \delta$) where δ is a delay after which this timer expires. The `RESET` action sets the timer to *off*.

With each timer there are associated a pseudo-signal and an implicit transition, called a timeout transition. When a timer expires, its timeout transition becomes enabled and may occur. The execution of the timeout transition, captured by the Expiration rule, adds the corresponding pseudo-signal to the process queue and reset the timer to *off*. If a `SET` or `RESET` operation is performed on an expired timer after adding the timer signal to the process queue (before the signal is consumed from the queue), the timer signal is removed from the queue. We write $\pi_t(q)$ for the queue obtained from q by projecting out the timeout signal t.

The *global* transition semantics of a specification *Spec* is given by a standard product construction: configurations are paired and global transitions synchronize via their common labels. The global step relation $\longrightarrow_\lambda \subseteq \Gamma \times \mathit{Label} \times \Gamma$ is given by the rules of table 2. Asynchronous communication between the two processes uses a signal s to exchange a common value v, as given by rule Comm. As far as τ-steps and communication messages using signals from the environment (Sig_{ext}) are concerned, each process can proceed on its own by rule Interleave. Both rules have a symmetric counterpart, which is omitted.

Time elapses by counting up the actual system time represented by `NOW`. Though there is no standardized time semantics in SDL, there exist two semantics widely accepted in the SDL-community [6]. According to one of them (the one, which is supported by the commercial SDL-design tools [15,13] and the one we work with), the transitions of SDL processes are instantaneous (take

Table 2. Parallel composition

$$\frac{\gamma_1 \rightarrow_{P!s(v)} \hat{\gamma}_1 \qquad \gamma_2 \longrightarrow_{P?s(v)} \hat{\gamma}_2 \qquad s \notin \mathit{Sig}_{ext}}{\gamma_1 \times \gamma_2 \rightarrow_\tau \hat{\gamma}_1 \times \hat{\gamma}_2}\ \textsc{Comm}$$

$$\frac{\gamma_1 \rightarrow_\lambda \hat{\gamma}_1 \qquad \lambda = \{\tau, P?s(v), P!s(v) \mid s \in \mathit{Sig}_{ext}\}}{\gamma_1 \times \gamma_2 \rightarrow_\lambda \hat{\gamma}_1 \times \gamma_2}\ \textsc{Interleave}$$

$$\frac{\mathit{blocked}(\gamma)}{\mathtt{NOW} \rightarrow_{tick} \mathtt{NOW}+1}\ \textsc{Tick}$$

zero time), time can only progress if the system is blocked: all the processes are waiting for further input signals (i.e., all input queues are empty, except for saved signals, and there is no `NONE` input enabled). TICK expresses it by the predicate *blocked* on configurations: $\mathit{blocked}(\gamma)$ holds if no move is possible in the system except a clock-tick, i.e., if $\gamma \rightarrow_\lambda$ for some label λ, then $\lambda = \mathit{tick}$. In other words, the time-elapsing steps are those with *least priority.* Due to the discarding feature, $\mathit{blocked}(\gamma)$ implies then $q = \epsilon$.

Later on, we refer to a segment of time separated by the time increment transitions as a *time slice.* (Note that time progression is discretised.) When the system time gets equal to some timer value, the timeout transition becomes enabled (EXPIRATION) and it can be executed at any point of the time slice. The time slice always starts with a firing of one of the enabled timeout transitions. This action unblocks the system. In case several timeout transitions become enabled at the same time, one of them is taken (non-deterministically) to unblock the system and the rest are taken later at any point of the time slice since they have the same priority as normal transitions.

Though complicated, such a time semantics is suitable for implementation purposes [11]. It is natural to model a timer as a unit advancing from the current moment of time derived by evaluation of `NOW` expression to the the time point specified by the expression ($\mathtt{NOW}+\delta$), i.e., waiting for a point of the system time.

3 Timer Transformation

A configuration of an SDL process is described by its current location, the valuations of timers and variables, and the content of the input queue. Since `NOW` gives an access to the current system time, executing $\mathtt{SET}(\mathtt{NOW}+\mathit{exp},\ t)$ with different (infinitely growing) values of `NOW`, we get different process configurations. Moreover, a timeout transition can add a timer signal at any point of the time slice. This blows up the state space due to the number of possible interleaving sequences of events. And furthermore, keeping a timeout signal in a process queue adds to the length of the state vector. Therefore, the way of modelling timers in SDL is not quite suitable for the model-checking purposes. In this section, we solve this problem by transforming SDL specifications, replacing a concept of timers with a new one.

3.1 Transformation Rules

To avoid the state-space explosion due to the interpretation of timers and the overhead caused by the management of timeout signals, we substitute the SDL concept of timers as a special kind of signals by a concept of timers as a special data type. A timer variable t represents declared in the original specification timer t. The value of this variable shows the delay left until the timer expiration. Since delays are non-negative, we use -1 to represent inactive timers. Therefore, the $\mathtt{RESET}(t)$ operation is transformed into the assignment of the -1 value to a timer variable t (Table 3, rule RESET TO ASSIGNMENT III) and the $\mathtt{SET}(\mathtt{NOW}+$

Table 3. Transformation Rules

$$\frac{l \longrightarrow_{g\,\triangleright ?t} \hat{l} \in Edg \qquad t \in Sig_{timeout}}{l \longrightarrow_{[g \bigwedge (t=0)]\,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true)\,\triangleright\, t:=-1} \hat{l} \in Edg} \text{ TI}\textsc{nput to} \text{ T}\textsc{imeout}\text{I}$$

$$\frac{l \longrightarrow_{g\,\triangleright\, \mathtt{SET(NOW}+exp,t\mathtt{)}} \hat{l} \in Edg}{l \longrightarrow_{[g \bigwedge (exp\, \geq\, 0)]\,\triangleright\, t:=exp} \hat{l} \in Edg} \text{ S}\textsc{et to} \text{ A}\textsc{ssignment}\text{I}$$

$$\frac{l \longrightarrow_{g\,\triangleright\, \mathtt{SET(NOW}+exp,t\mathtt{)}} \hat{l} \in Edg}{l \longrightarrow_{[g \bigwedge (exp<0)]\,\triangleright\, t:=0} \hat{l} \in Edg} \text{ S}\textsc{et to} \text{ A}\textsc{ssignment}\text{II}$$

$$\frac{l \longrightarrow_{g\,\triangleright\, \mathtt{RESET}(t)} \hat{l} \in Edg}{l \longrightarrow_{g\,\triangleright\, t:=-1} \hat{l} \in Edg} \text{ R}\textsc{eset to} \text{ A}\textsc{ssignment}\text{III}$$

$$\frac{l \longrightarrow_{g\,\triangleright ?s(x)} \hat{l} \in Edg \Rightarrow \ s \neq t \qquad t \in Sig_{timeout}}{l \longrightarrow_{(t=0)\,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true)\,\triangleright\, t:=-1} l \in Edg} \text{ TD}\textsc{iscard to} \text{ T}\textsc{imeout}\text{II}$$

$exp, t)$ operation is transformed into the assignment of exp in case $exp \geq 0$ (Table 3, Set to AssignmentI), or the assignment of 0 in case $exp < 0$ (Table 3, Set to AssignmentII).

In the original system, a timer whose value in the original system is less than or equal to the current system time may expire. The transformed system should demonstrate the same behaviour. Since we suppose the value of a transformed timer to be a delay left until its expiration, only the timers whose values are equal to 0 may expire. Therefore, we replace inputs of timeout signals by the `NONE` input guarded by the $(t = 0)$ condition, followed by timer deactivation (Table 3, rule TInput to TimeoutI). A possible discard of a timeout signal is imitated by analogous actions (Table 3, rule TDiscard to TimeoutII). Fig. 1 illustrates the application of the transformation rules.

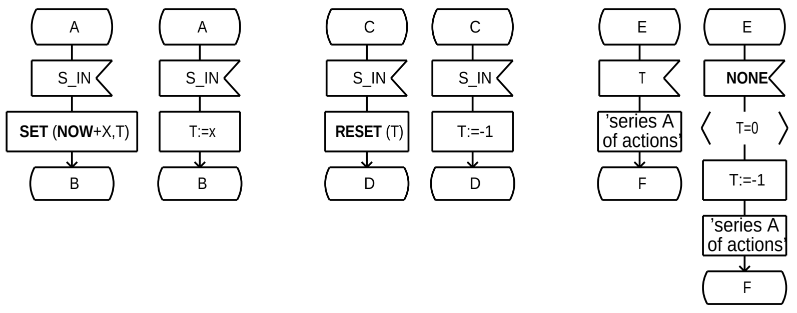


Fig. 1. Transformation scheme

Table 4. Step semantics for transformed specification

$$\frac{l \longrightarrow_{g \,\triangleright\, x \,:=\, exp} \hat{l} \in Edg \qquad [\![g]\!]_\vartheta = true \qquad [\![exp]\!]_\vartheta = v \qquad x \in TVar}{(l, \vartheta, \varphi, q) \rightarrow_\tau (\hat{l}, \vartheta, \varphi[x \mapsto v], q)} \text{ TASSIGN}$$

$$\frac{blocked(l, \vartheta, \varphi, q)}{(l, \vartheta, \varphi, q) \rightarrow_{tick} (l, \vartheta, \varphi[t \mapsto dec(t)], q)} \text{ TICK}$$

Table 4 gives the modifications of the step semantics caused by the timer transformation. TASSIGN defines the semantics of assignments for the timer variables. The rules INPUT, DISCARD, ASSIGN, OUTPUT, RECEIVE, COMM and INTERLEAVE coincide with ones of Table 1, and, therefore, omitted. Note that the system time is not present in the transformed system — one infinitely growing variable would be enough to cause the state-space explosion. Instead of increasing the system time, the tick transition (rule TICK of Table 4) decreases the values of timer variables. Like the TICK transition of the original system, this transition can take place only if the system is blocked, and "blocked" has exactly the same meaning as before. The *dec* operation decreases all the positive values of timer variables by 1 and leaves the others unchanged.

Note that the transformation rules are developed under the assumption that `NOW` operator appears in the system in the scope of `SET` operations only, and all the `SET` operations are of the form `SET`$(\texttt{NOW} + \delta, t)$. In the sequel, we consider the systems of this type. In case `NOW` is employed in a specification for measuring time intervals between, these fragments of the specification can be respecified with the use of an additional timer variable.

3.2 Model Equivalence

The transformed system does not give a straightforward reflection of the original system behaviour. While the actions that are not related to timers are left unchanged, sendings of timeout signals are projected out, and consumptions of timeout signals from the process queue are mimicked by the corresponding `NONE` inputs, whose enabling conditions are guaranteed to be true in this case. Such a projection is not harmful from the verification point of view because not the presence or absence of a timeout signal but the consumption of it and the choice of the following actions are important. The same concerns process queues: saying that the content of some queue in the transformed system is the same as the content of the corresponding queue in the original system, we mean that the projections of the queues on $Sig \setminus Sig_{timeout}$ coincide. The main requirement imposed on the configurations is that the valuations of variables should equal:

Definition 1 (relation $\approx$ on configurations). *Let $\gamma = (l, \theta, \phi, q)$ and $\gamma' = (l', \theta', \varphi, q')$ be configurations of processes P and P'. We write $\gamma \approx \gamma'$ iff for any $x \in Var$: $[\![x]\!]_\theta \; = \; [\![x]\!]_{\theta'}$.*

Definition 2 (simulation). *Let P and P' be two processes with sets of configurations Γ and Γ' and $R \subseteq \Gamma \times \Gamma'$ a relation on configurations. R is a* simulation *iff $R \subseteq \approx$, and ($\gamma R \gamma'$ and $\gamma \to_\lambda \hat{\gamma}$) implies that at least one of the following conditions holds:*

1. *$\gamma' \to_\lambda \hat{\gamma}'$ and $\hat{\gamma} R \hat{\gamma}'$, for some configuration $\hat{\gamma}'$;*
2. *$\lambda = \tau$ and $\hat{\gamma} R \gamma'$;*
3. *$\gamma' \to_\tau \gamma'_1 \to_\tau \ldots \to_\tau \gamma'_n \to_\lambda \hat{\gamma}'$ for some $n \geq 0$ such that $\hat{\gamma} R \hat{\gamma}'$ and $\gamma R \gamma'_i$ for all γ'_i.*

We write $P \preceq P'$, if there exists a simulation relation R such that $\gamma_0 R \gamma'_0$ for the initial configurations γ_0 and γ'_0 of P and P' respectively.

The definition of simulation is analogously used for systems.

Lemma 1. *For all formulas φ from next-free LTL mentioning only variables from Var, $S \preceq S'$ and $S' \models \varphi$ implies $S \models \varphi$.*

Definition 3. *Let P' be a process derived from a process P using the transformation rules defined in Table 3, $\gamma = (l, \theta, \phi, q)$ and $\gamma' = (l', \theta', \varphi, q')$ be their configurations and* `NOW` *be the system time related to process P. We say that $(\gamma, \texttt{NOW}) Q \gamma'$ iff the following conditions are satisfied:*

1. *$\gamma \approx \gamma'$;*
2. *$q' = \pi_{Sig_{timeout}}(q)$, where $\pi_{Sig_{timeout}}(q)$ is obtained from q by projecting out the signals from $Sig_{timeout}$;*
3. *if $[\![t]\!]_\phi = on(v)$ and $[\![t]\!]_\varphi = k$ then $k + [\![\texttt{NOW}]\!] = \max\{[\![\texttt{NOW}]\!], v\}$;*
4. *if $[\![t]\!]_\phi = off$ and a timeout signal t is not in q then $[\![t]\!]_\varphi = -1$;*
5. *if $[\![t]\!]_\phi = off$ and a timeout signal t is in q then $[\![t]\!]_\varphi = 0$.*

This relation Q can be lifted up to systems.

Lemma 2. *Let S' be a system derived from a system S according to the transformation rules defined in Table 3, γ and γ' be configurations of S and S' respectively, and $\gamma Q \gamma'$. Then $blocked(\gamma)$ iff $blocked(\gamma')$.*

Theorem 1 (simulation). *Let S and S' be systems defined as above. Then Q is the simulation relation on their configurations, $S \preceq S'$.*

Proof sketch. It is straightforward to check on the rules of Tables 1, 3, 4 that for single processes $P \preceq P'$. There, for the case of Expiration in P take no step in P', for the case of TInput take TimeoutI, and for the case of TDiscard take TimeoutII. For systems, prove that $P_1 \preceq P'_1$ and $P_2 \preceq P'_2$ implies $P_1 \parallel P_2 \preceq P'_1 \parallel P'_2$, proceeding similarly by case analysis on the rules of Table 2, using Lemma 2. □

Ideally, we would like Q^{-1} to be a simulation relation as well, and establish by that a bisimulation between S and S'. However, it is not the case—an attempt to establish a simulation in the reverse direction fails while considering the Timeout case in the transformed system. In principle, Timeout should be mimicked by taking Expiration and TInput. But the Expiration step in the original system cannot be taken earlier than the decision about Timeout is taken in the transformed system. On the other hand, when Timeout is taken, it could be already too late to take the Expiration step, since the process queue of the original system can be non-empty, which would mean that the timeout signal could not be immediately consumed from the process queue. We solve this problem by proving the trace inclusion for S and S'.

Definition 4 (trace). *Let N be a set $\{0, 1, 2, \ldots, n\}$. We say that a pair of mappings $\sigma = (\sigma_\gamma, \sigma_\lambda)$ with $\sigma_\gamma : N \longrightarrow \Gamma$ and $\sigma_\lambda : N \setminus \{0\} \longrightarrow Label$ is a trace of length n generated by a system S with a set of configurations Γ iff for any $i : 0 \leq i < n$, $\sigma_\gamma(i) \rightarrow_{\sigma_\lambda(i+1)} \sigma_\gamma(i+1)$ is a step of S.*

In case $N = \mathbb{N}$ we say that σ is an infinite trace.

Definition 5 (equivalence of traces). *Let σ and σ' be traces of length n, n' generated by systems S and S' resp. We say that $\sigma \equiv_{tr} \sigma'$ iff there exists $U \subseteq N \times N'$ such that*

1. *$(i, j) \in U$ implies $\sigma_\gamma(i) \approx \sigma'_\gamma(j)$.*
2. *For any $i : 0 \leq i \leq n$, there exists $j : 0 \leq j \leq n'$ such that $(i, j) \in U$.*
3. *For any $j : 0 \leq j \leq n'$, there exists $i : 0 \leq i \leq n$ such that $(i, j) \in U$.*
4. *For any $i, j : 0 \leq i \leq n, 0 \leq j \leq n'$, $(i, j) \in U$ implies that one of the following conditions holds:*
   - $\sigma_\lambda(i+1) = \sigma'_\lambda(j+1) \wedge (\sigma_\gamma(i+1), \sigma'_\gamma(j+1)) \in U \wedge$ $(\forall k : 0 \leq k < i+1 : (k, j+1) \notin U) \wedge (\forall k : 0 \leq k < j+1 : (i+1, k) \notin U)$;
   - $\sigma_\lambda(i+1) = \tau \wedge (\sigma_\gamma(i+1), \sigma'_\gamma(j)) \in U \wedge (\forall k : 0 \leq k < j : (i+1, k) \notin U)$;
   - $\sigma_\lambda(j+1) = \tau \wedge (\sigma_\gamma(i), \sigma'_\gamma(j+1)) \in U \wedge (\forall k : 0 \leq k < i : (k, j+1) \notin U)$.

We write $S' \preceq_{tr} S$ iff for every trace σ' of S' there exits a trace σ in S such that $\sigma \equiv_{tr} \sigma'$.

Lemma 3. *For all formulas φ from next-free LTL mentioning only variables from Var, $S' \preceq_{tr} S$ and $S \models \varphi$ implies $S' \models \varphi$.*

Theorem 2 (trace inclusion). *Let S' be a system derived from a system S according to the transformation rules defined in Table 3. Then $S' \preceq_{tr} S$.*

Proof sketch. For an arbitrary trace σ' of S', we are constructing an equivalent trace in S. We impose an additional condition on U that $(i, j) \in U$ implies that $\sigma_\gamma(i) Q \sigma'_\gamma(j)$. Then construct a trace σ of S and a relation U between traces by induction on the length of σ', applying the stepwise simulating of σ' with a special treatment of the Timeout case. For Timeout, insert the Expiration step at the appropriate place in the constructed trace, modifying U respectively (by shifting the tail of σ in U), and then take TInput or TDiscard respectively. □

Corollary 1. *Let S and S' be systems defined as above, and φ a next-free LTL-formula mentioning only variables from Var. Then $S \models \varphi$ iff $S' \models \varphi$.*

4 Timer Abstraction

Abstraction, intuitively, means replacing one semantical model by an abstract, in general, simpler one. In addition to the requirement that an abstract (verification) model should have a smaller state space than the concrete (implementation) one, the abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well[2]. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [4,5]. The requirement that Abstract Interpretation puts on the relation between the concrete model and its safe abstraction can be formalized as a requirement on the relation between the data operations of the concrete system and their abstract counterparts as follows. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value which, intuitively, "describes" the concrete value. On the other hand, the concretisation function γ maps each abstract value to a set of concrete values that are described by the abstract value. Abstract states are ordered according to their precision: $x^\alpha \preceq y^\alpha \Leftrightarrow \gamma(x^\alpha) \subseteq \gamma(y^\alpha)$. For every operation (function) f_{conc} on the concrete state space, an abstraction f_{abs} needs to be defined which "mimics" f_{conc}. In general, the abstraction can be nondeterministic. This is formally captured by letting f_{abs} be a function into the powerset over the domain of abstract values. The requirement of mimicking is then formally phrased as:

$$\forall x \in Val\ \exists y \in f_{abs}(\alpha(x)) : \alpha(f_{conc}(x)) \preceq y$$

Further we call this the *safety statement.*

Besides decreasing the state space of the system and safety, the main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that an abstraction captures all essential points in the system behaviour, i.e., it is not "too abstract". The safe abstraction of timers for Promela translations of SDL-specification from [2] does not meet this requirement well enough.

The abstraction in [2] is based on a natural idea of allowing timers to expire at an arbitrary moment after they are set. A typical problem arising when one starts to apply this abstraction in practice is introducing zero-time cycles which are not present in the concrete model. A usual pattern for SDL-specifications is that a timer schedules some periodical activity; after a timer-out signal is consumed by a process, some actions are taken, the timer is set again, and the process returns to the same control state where it was before the consumption of the timer signal. Since the abstraction allows a timer to expire at any arbitrary moment after its setting (also immediately after it has been set), an undesirable

[2] A safe abstract system is, intuitively, a system whose behaviour (the set of all transitions) is a superset of the concrete system behaviour.

cyclic behaviour can be introduced. The “timeout input – timer setting – timer expiration” chain of transitions can be executed infinitely many times and all the other behavioural branches may be ignored forever. Therefore, the properties which hold for the concrete model and which are expected to hold independently of the timer settings, fails to hold for the abstract model since “independently” means in this case that the property holds for the concrete model whatever *positive* delay is assigned to a timer. Another problem arises in case a timer serves as a guard preventing from taking a transition too early. With abstracting time, this timer guard is broken.

We propose an abstraction for timers that keeps this guard delaying the timer expiration. A concrete timer that can be set only to delays exceeding some k is abstracted according to the rules given in Table 5. The setting of the concrete timer to a $(\mathtt{NOW} + x)$ value is replaced with setting it to $(\mathtt{NOW} + k)$ (assuming $x \geq k$ to be an invariant for the setting operations), and the timer is allowed to expire at any point of time after the delay of k time units (see Fig. 2). Varying k, we can change the refinement degree of the abstraction. Taking k equal to 0, we get the most abstract version of it, which is an abstraction from [2] where not just timers but time is abstracted. Taking k equal to the lower boundary of the timer delays in the system, we get the most refined abstraction that can be obtained with this abstraction schema.

Table 5. Abstract SET and INPUT operations

$$\frac{l \longrightarrow_{g \,\triangleright\, \mathtt{SET(NOW}+x,t)} \hat{l} \in Edg \qquad [\![x]\!]_{\geq k}}{l \longrightarrow_{g \,\triangleright\, \mathtt{SET(NOW}+k,t)} \hat{l} \in Edg}\ \textsc{Set}$$

$$\frac{l \longrightarrow_{g \,\triangleright ?t} \hat{l} \in Edg \qquad t \in Sig_{timeout}}{l \longrightarrow_{g \,\triangleright ?t} \hat{l} \in Edg}\ \textsc{ExpireNow}$$

$$\frac{l \longrightarrow_{g \,\triangleright ?t} \hat{l} \in Edg \qquad t \in Sig_{timeout}}{l \longrightarrow_{g \,\triangleright ?t} \longrightarrow_{(true) \,\triangleright\, \mathtt{SET(NOW}+1,t)} l \in Edg}\ \textsc{ExpireLater}$$

$$\frac{l \longrightarrow_{g \,\triangleright ?s(x)} \hat{l} \in Edg \Rightarrow s \neq t \qquad l \in Loc_i \qquad t \in Sig_{timeout}}{l \longrightarrow_{g \,\triangleright ?t} l \in Edg}\ \textsc{DiscardNow}$$

$$\frac{l \longrightarrow_{g \,\triangleright ?s(x)} \hat{l} \in Edg \Rightarrow s \neq t \qquad l \in Loc_i \qquad t \in Sig_{timeout}}{l \longrightarrow_{g \,\triangleright ?t} \longrightarrow_{(true) \,\triangleright\, \mathtt{SET(NOW}+1,t)} l \in Edg}\ \textsc{DiscardLater}$$

Now we shall show that this abstraction only adds behaviour while preserving all the behaviour of the concrete system. Since the timer semantics described in Section 3 is simpler than the original SDL semantics, it is easier to prove safety of the abstraction w.r.t. transformed systems. Table 6 gives the abstraction rules for the transformed system S' derived by applying the transformation to the

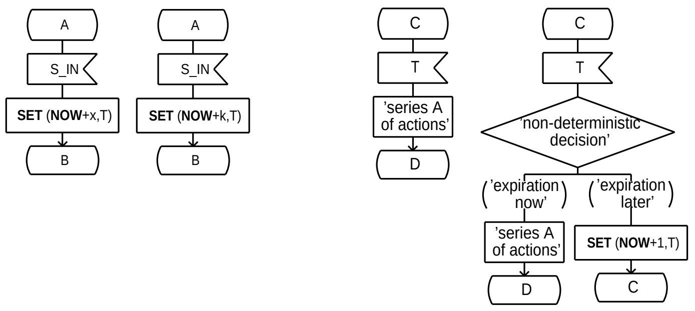


Fig. 2. Abstraction scheme

abstraction rules from Table 5. Let S be a concrete system and ${}_{\alpha}S$ its abstraction. Consider systems S' and ${}_{\alpha}S'$ derived from S and ${}_{\alpha}S$ using the transformation rules from Table 3. Then (due to Corollary 1) we have the property preservation directions from S to S' and from ${}_{\alpha}S'$ to ${}_{\alpha}S$. So it remains to show that ${}_{\alpha}S'$ is a safe abstraction of S'.

In order to distinguish the values of timer variables in the abstract system from their values in the concrete one, we write $1^{\alpha}, 2^{\alpha}, \ldots, k^{\alpha}$ for the abstract values. The abstraction function α and the concretisation function γ are then as follows:

$$\alpha(x) = \begin{cases} k^{\alpha} & \text{if } x \geq k, \\ x^{\alpha} & \text{if } 0 \leq x < k, \\ -1^{\alpha} & \text{if } x = -1; \end{cases} \qquad \gamma(x^{\alpha}) = \begin{cases} \{y \,|\, y \geq x\} & \text{if } 0^{\alpha} \leq x^{\alpha} \leq k^{\alpha}, \\ \{-1\} & \text{if } x^{\alpha} = -1. \end{cases}$$

Hence, we have $0^{\alpha} \succeq 1^{\alpha} \succeq \ldots \succeq k^{\alpha}$.

Lemma 4. *The abstraction defined by the rules of Table 6 is safe.*

Proof sketch. It is straightforward to check on the rules of Table 6 that the safety statement is not violated.

Lemma 4 implies that we have the property preservation in the direction from ${}_{\alpha}S'$ to S', which immediately gives the following conclusion:

Theorem 3. *Let ${}_{\alpha}S$ be a system derived from a system S using the abstraction rules given in Table 5, φ a next-free LTL-formula mentioning only variables from Var, and $S' \models \varphi$. Then $S \models \varphi$.*

The verification experiments showed that the smallest number of states is usually obtained with abstractions where k is equal to 1, not 0. (We compare the number of states in the models, to which the transformation from Section 3 is applied.) The state space normally grows with increasing k, which is no wonder since the number of possible states of the timer itself grows in that case. Rather unexpected was the fact that taking k equal to 0 increases the state space and can lead to the state-space explosion. The behaviour of the system with zero timer delay often has a different nature of regularity, the behaviour branches excluded formerly by the timer guards can be taken, which explains this phenomenon.

Table 6. Abstraction rules for a transformed system

$$\frac{l \longrightarrow_{g \,\triangleright\, t:=x} \hat{l} \in Edg \qquad [\![x]\!]_{\varphi} \geq k}{l \longrightarrow_{g \,\triangleright\, t:=k} \hat{l} \in Edg}\ \textsc{TAssignI}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, t:=x} \hat{l} \in Edg \qquad 0 \leq [\![x]\!]_{\varphi} < k}{l \longrightarrow_{g \,\triangleright\, t:=x} \hat{l} \in Edg}\ \textsc{TAssignII}$$

$$\frac{l \longrightarrow_{[g \,\bigwedge (t=0)] \,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true) \,\triangleright\, t:=-1} \hat{l} \in Edg}{l \longrightarrow_{[g \,\bigwedge (t=0)] \,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true) \,\triangleright\, t:=-1^{\alpha}} \hat{l} \in Edg}\ \textsc{TimeoutNow}$$

$$\frac{l \longrightarrow_{[g \,\bigwedge (t=0)] \,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true) \,\triangleright\, t:=-1} \hat{l} \in Edg}{l \longrightarrow_{[g \,\bigwedge (t=0)] \,\triangleright ?\mathtt{NONE}} \longrightarrow_{(true) \,\triangleright\, t:=1} l \in Edg}\ \textsc{TimeoutLater}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, t:=-1} \hat{l} \in Edg}{l \longrightarrow_{g \,\triangleright\, t:=-1^{\alpha}} \hat{l} \in Edg}\ \textsc{TAssignIII}$$

5 Conclusion

The proposed transformation of SDL-timers is a simple and inexpensive step that can be considered as a zero-phase of the translation of SDL-specifications to a model-checker input language. The transformation is aimed at reducing the state space to a finite domain. A side issue (though important on its own) is that the described transformation gives a different insight into the timer semantics. Our experience shows that the complicated time semantics of SDL can lead to errors due to its misunderstanding both in the design-phase and in the translation-phase (cf. [2,12]). Treatment of timers as variables is simpler than treatment them as signals. Moreover, the transformation removes the concept of global time, whose existence does not look realistic for many sorts of distributed systems.

The abstraction of timers is aimed at the state space reduction. Due to its flexibility, it is applicable to a wider range of problems that the earlier used for SDL timer abstractions. Its practical application confirmed its efficiency for the real-life situations (see [12]).

References

1. D. Bošnački, D. Dams, *Integrating Real Time into Spin: A Prototype Implementation*, S. Budkowski, A. Cavalli, E. Najm, editors, Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98), Kluwer, 1998.
2. D. Bošnački, D. Dams, L. Holenderski, N. Sidorova, *Verifying SDL in Spin*, Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000, LNCS 1785, pp. 363-377, Springer, 2000.

3. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
4. P. Cousot, R. Cousot, *Abstract Interpretaion: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, In the 4th POPL, Los Angeles, CA, ACM, January 1977.
5. D. Dams, *Abstract Interpretation and Partition Renement for Model Checking*, PhD thesis, Eindhoven University of Technology, July 1996.
6. U. Hinkel, *Verification of SDL Specifications on the Basis of Stream Semantics*, In Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC, Y. Lahav, A. Wolisz, J. Fischer, E. Holz (eds.), Humboldt-Universitaet zu Berlin.
7. G. J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991.
8. G.J. Holzmann, J. Patti, *Validating SDL Specification: an Experiment*, In E. Brinksma, G. Scollo, Ch.A. Vissers, editors, Protocol Specification, Testing and Verification, Enchede, The Netherlands, 6-9 June 1989, pp. 317-326, Amsterdam, North-Holland, 1990.
9. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, *Property Preserving Abstractions for the Verification of Concurrent Systems*, In Formal Methods in System Design, Kluwer Academic Publ., 6, 1-36, 1995.
10. F. Regensburger, A. Barnard, *Formal Verification of SDL Systems at the Siemens Mobile Phone Department*, In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98) 1998, LNCS 1384, pp. 439-455, Springer, 1998.
11. A. Olsen *et al.*, *System Engineering Using SDL-92*, Elsevier Science, North-Holland, 1997.
12. N. Sidorova, M. Steffen, *Verifying Large SDL-Specifications using Model Checking*, In Proc. of 10th International SDL-Forum, Copenhagen, Denmark, 2001, LNCS 2078, pp. 403-420, Springer, 2001.
13. Telelogic Malmö AB. *SDT 3.1 User Guide, SDT 3.1 Reference Manual*, Telelogic, 1997.
14. H. Tuominen, Embedding a Dialect of SDL in PROMELA, 6th Int. SPIN Workshop, LNCS 1680, pp. 245-260, Springer, 1999.
15. Verilog, *ObjectGEODE SDL Simulator - Reference Manual*, 1996.

Accurate Widenings and Boundedness Properties of Timed Systems

Supratik Mukhopadhyay and Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{supratik|podelski}@mpi-sb.mpg.de

Abstract. We propose a symbolic model checking procedure for timed systems that is based on operations on constraints. To accelerate the termination of the model checking procedure, we define history-dependent widening operators, again in terms of constraint-based operations. We show that these widenings are accurate, i.e., they don't lose precision even with respect to the test of boundedness properties.

1 Introduction

For the last ten years, the verification problem for timed systems has received a lot of attention (see e.g., [2,3,12,22,24]). The problem has been shown to be decidable in [2]. Most of the verification approaches to this problem have been based either on a region graph, which is a finite quotient of the infinite state graph, or on some variants of it (that use convex/non-convex polyhedra and avoid explicit construction of the full graph). But, as we show below, region-graph based approaches (or its variants) cannot be used for dealing with *boundedness* (unboundedness) properties. This is due to the fact that the partitioning of the state space induced by the region equivalence (or any other technique that takes into account the maximal constant in the guards) is guaranteed to be *pre-stable* but may not be *post-stable*[1].

A boundedness property is of the form $\exists k \geq 0\ AG(x \leq k)$ where x is a clock (read this specification as: there exists a nonnegative k such that for all paths starting from the initial position, the value of the clock x does not exceed k throughout the path). An unboundedness property is the dual of a boundedness property: $\forall k \geq 0\ EF(x > k)$. These properties are useful in verification because if the designer knows that the value of a clock should never exceed a constant, then satisfaction of an unboundedness property by the design immediately informs the designer of a possible bug in the design. Also, the implementor can use this information to save some hardware while implementing the design (in hardware). Obtaining such information usually requires lot of insight.

Consider the timed automaton (see [2] for a definition of timed automata) given in Figure 1 (it has two clocks x and y and four locations 0, 1, 2 and 3; the

[1] the definitions of pre-stable and post-stable partitions can be found in [1].

guards and resets for the edges are indicated at the top of or beside the edges; the invariants of the locations are indicated above or below the locations). Let us try to see whether the system satisfies the property $\exists k \geq 0\ AG(x \leq k)$, where the clock x in the formula refers to the clock x in the automaton. If we use the region graph technique, we will see that the regions (the maximal constant is 2 here) $(1 < y < 2, x > 2)$, $(y = 1, x > 2)$ and $(0 < y < 1, x > 2)$ (we do not enumerate all the reachable regions; the variables x,y range over reals) are reachable. One may now conclude, on the basis of this reachability analysis, that the automaton does not satisfy the above boundedness property (note that all the three regions given above are unbounded). Unfortunately, this is not true; the value of the clock x never exceeds 6 (just six)! Region graphs (or its variants) cannot be directly used for model checking for boundedness properties!

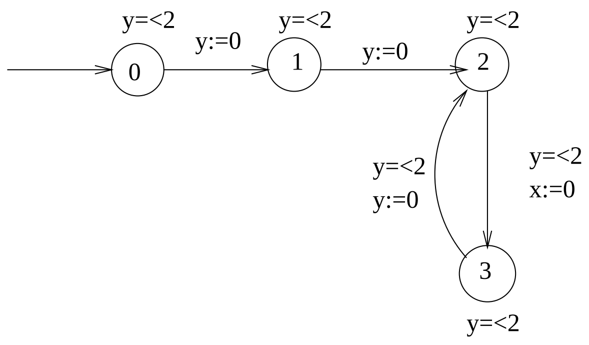


Fig. 1. Illustrating unboundedness (boundedness) property

Now consider a reachability analysis for this timed automaton using the algorithm in Figure 9 (this algorithm is a well-known simple symbolic forward reachability analysis algorithm; due to lack of space it is provided in the Appendix). The algorithm terminates generating the following set of reachable states: $\langle l\_0(x,y), x = 0, y = 0\rangle$, $\langle l\_0(x,y), x = y, y \geq 0, y \leq 2\rangle$, $\langle l\_1(x,y), 0 \leq x \leq 2, y = 0\rangle$, $\langle l\_1(x,y), x - y \geq 0, y - x \geq -2, y \geq 0, y \leq 2\rangle$, $\langle l\_2(x,y), 0 \leq x \leq 4, y = 0\rangle$, $\langle l\_2(x,y), x - y \geq 0, y - x \geq -4, y \geq 0, y \leq 2\rangle$, $\langle l\_3(x,y), 0 \leq y \leq 2, x = 0\rangle$ and $\langle l\_3(x,y), x - y \geq 0, y \leq 2, x \geq 0\rangle$ (the states are tuples of locations and constraint stores; we write l_i for the location i; symbolic forward reachability analysis for the timed automaton in Figure 1 produces these constraints). It can be easily found out from the set of reachable states (by projecting the constraints on the x-axis) that the value of the clock x never goes beyond 6 and hence the above boundedness property is satisfied.

It can be shown that if the (symbolic) model checking algorithm in Figure 9 terminates, we can successfully model check for boundedness (unboundedness) properties. It is now natural to ask the question whether the procedure in Figure 9 is guaranteed to terminate. The answer is 'no'; consider the timed automaton in Figure 2 — the algorithm in Figure 9 will not terminate for this example (an infinite sequence of "states" which are not "included" in the "previously" generated states are produced). Of course, the procedure can be forced to terminate by including some maximal constant manipulation techniques (as the trim operation introduced in [23] or the extrapolation operation [12] or the preprocessing step [17]). But then, like the region graph technique, it can be shown that these techniques cannot be directly used for model checking for boundedness properties. So the natural thing now would be to develop techniques that force the termination of the procedure in Figure 9 (in cases where it is possible) but do not lose any information with respect to boundedness properties. It is in this context that *history-dependent constraint widenings* come into play.

Before introducing our framework of history-dependent constraint widenings (accurate widenings), let us try to see whether the already-existing *abstract interpretation* framework [10] can provide solutions to the problems described above. Abstraction interpretation techniques [10] are useful tools to force termination of the symbolic model checking procedures. Here one obtains a semi-test by introducing abstractions that yield a conservative approximation of the original property. Such methods have been successfully applied to many nontrivial examples [12,3,24,19]. While these abstractions force the termination of the model checking procedure, they sacrifice their accuracy in the process (note that by accuracy, we mean not only accuracy with respect to reachability properties, but also with respect to boundedness properties). One of the most commonly used abstractions is the *convex hull* abstraction [24,12,3].

The application of automated, application independent abstractions that enforce termination, as is done in program analysis, to model checking seems difficult for the reason that the abstractions are often too rough[2]. To know the accuracy of an abstraction is important both conceptually and pragmatically. As Wong-Toi observes in [24],

> ...The approximation algorithm proposed is clearly a heuristic. It would be of tremendous value to have analytical arguments for when it would perform well, for when it would not....

As we saw above, any symbolic model checking procedure that "loses" accuracy will not be able to model check for boundedness (unboundedness) properties. Hence, in this paper, we propose a framework, to provide a partial answer to the question asked by Wong-Toi, viz., to determine automatically (using analytical methods) whether an abstraction performs well (does not lose accuracy) in a situation and then apply the abstraction.

We present methods that carry over the advantages of abstract interpretation techniques without losing precision. To be more specific, we apply history-

[2] Note the statement of Halbwachs in [13], that "Any widening operator is chosen under the assumption that the program behaves regularly. . . . Now the assumption of regularity is obviously abusive in one case: when a path in the loop becomes possible at step n, the effect of this path is obviously out of the scope of extrapolation before step n (since the actions performed on this path have never been taken into account). . . "

dependent constraint widening techniques, as already foreseen in [10,11], to provide an application-independent abstract interpretation framework for model checking for timed systems. Basing our intuitions on techniques from Constraint Databases [21], we show that abstractions of the model checking fixpoint operator, through a set of widening rules, can yield an accurate model checking procedure. These abstractions are based on syntax of the constraints rather than their meaning (the solution space) in contrast with previous approaches (e.g., [3, 19,24,4]). As we demonstrate on examples, they can drastically reduce the number of iterations or even, in some cases, force termination of an otherwise non-terminating test. In contrast with the abstract interpretation techniques used for program analysis, they do not always force termination; instead their abstraction is accurate. That is, they do not lose information with respect to the original property; when they terminate, they provide information which is sufficient even for model checking for boundedness (unboundedness) properties; i.e., in cases where termination is achieved, the abstractions are sound and complete. Also, being based on the syntax of the constraints they can be implemented efficiently (they do not require computation of the convex hull like [24,3,19];). We first show toy examples in which our abstractions (henceforth called widening rules) either achieve termination in an otherwise non-terminating analysis or drastically accelerate the termination of symbolic forward reachability analysis.[3] We then show the performance of a prototype model checker, implemented using the techniques presented in this paper, on some standard benchmark examples taken from literature. In the Conclusion, we discuss the generality of our approach. The proofs and details are omitted from this extended abstract for lack of space. We invite the reader to go through an extended version of this abstract available at http://www.mpi-sb.mpg.de/∼supratik/mainwide.ps.

2 Timed Automata, Constraints, and Model Checking

For the purposes of this paper, we model timed systems using timed automata. We refer the reader to [2] for a formal treatment of timed automata.

We now fix the formal set up of this paper. We use lower case Greek letters for a constraint and upper case Greek letters for a set of constraints (which stands for their disjunction). The interpretation domain for our constraints is $\mathcal{R}$ the set of reals. We write $\mathbf{x}$ for the tuple of variables $x_1, \ldots, x_n$ and $\mathbf{v}$ for the tuple of values $v_1, \ldots, v_n$. As usual, $\mathcal{R}, \mathbf{v} \models \varphi$ is the validity of the formula φ under the valuation $\mathbf{v}$ of the variables $x_1, \ldots, x_n$. We formally define the relation denoted by a constraint φ as:

$$[\varphi] = \{\mathbf{v} \mid \mathcal{R}, \mathbf{v} \models \varphi\}$$

[3] Note that we consider forward analysis, instead of backward analysis, for the obvious advantages mentioned in [18] (Forward analysis is amenable to on-the-fly local model checking and also to partial order reductions. These methods ensure that only the reachable portion of the state space is explored). Moreover, backward analysis cannot be used for model checking for boundedness properties.

Note that $x_1, \ldots, x_n$ act as the free variables of φ and implicitly all other variables are existentially quantified. We write $\varphi[\mathbf{x}']$ for the constraint obtained by alpha-renaming from φ. We define $[\Phi]$, the relation denoted by a set of constraints Φ with respect to variables $x_1, \ldots, x_n$ in the canonical way. For a constraint φ and a set of constraints $\{\psi_1, \ldots, \psi_k\}$, we write $\varphi \models \bigvee_{i=1}^{k} \psi_i$ iff $[\varphi] \subseteq \bigcup_{i=1}^{k} [\psi_i]$. For sets of constraints Φ_1 and Φ_2 (where by a set of constraints $\Phi = \{\varphi_i\}$, we mean $\bigvee_i \varphi_i$), we write $\Phi_1 \models \Phi_2$ if for all $\varphi \in \Phi_1$ there exists $\varphi' \in \Phi_2$ such that $[\varphi] \subseteq [\varphi']$.We write an event (an edge transition or a time transition or a composition of several edge and time transitions) as **cond** ψ **action** φ, where the guard ψ is a constraint over $x_1, \ldots, x_n$ and the action φ is a constraint over the variables $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$. The primed variable x' denotes the value of the variable x in the successor state. Note that we use interleaving semantics for our model. **We will use a set of constraints Φ to represent a set of states $\mathcal{S}$ if $\mathcal{S} = [\Phi]$**. The successor of a set of states of such a set with respect to an event $e \equiv$ **cond** ψ **action** φ is represented by the constraints obtained by conjoining the guard ψ and the action φ of each event with each constraint φ of Φ:

$$post_{|e}(\Phi) = \{\exists_{-\mathbf{x}'} \varphi \wedge \psi \wedge \varphi \mid \varphi \in \Phi, \mathcal{R} \models \varphi \wedge \psi \wedge \varphi\}$$

where the existential quantifier is over all variables but $\mathbf{x}'$.

We next formulate possibly non-terminating symbolic model checking procedures for boundedness properties, in our constraint-based framework, The template for the algorithm is given in Figure 9. Here $post(\Phi) = \cup_{e \in \mathcal{E}} post_{|e}(\Phi)$ where $\mathcal{E}$ is the set of all events of the timed system (*simple* and *compound*; see below for definitions of compound (composed) events). The algorithm is basically a (inflationary) fixpoint computation algorithm. Note that the template Symbolic-Boundedness can be used for model checking for the logic $\mathcal{L}_s$ [22]. Also note that the algorithm is breadth first. In the sequel, we call the algorithm Symbolic-Boundedness as the breadth first (symbolic forward) reachability analysis algorithm.

The locations of a timed automaton can be encoded as finite domain constraints (in our algorithms we assume that the locations are encoded as finite domain constraints). We denote a position (simply a state) [2,16] of the timed automaton having location component ℓ as $\ell(\mathbf{v})$ where $\mathbf{v}$ denotes the values of the clocks. In general, for a set $\mathcal{S}$ of states having the location component ℓ, we write $\langle \ell, S \rangle$, or $\langle \ell(\mathbf{x}), \varphi \rangle$, where φ is a constraint and $S = [\varphi] = \{\mathbf{v} \mid \ell(\mathbf{v}) \in \mathcal{S}\}$. Here the free variables of φ are $\{x_1, \ldots, x_n\}$. In the sequel, we will refer to a set of states with location component ℓ and represented by $\langle \ell(\mathbf{x}), \varphi \rangle$ as a symbolic state or simply a state when it is clear from context.

3 Widening Rules

In this section, we consider how one can achieve (or just speed up) termination of the breadth first forward reachability analysis algorithms for boundedness (as well as safety) properties. We define widening rules that are accurate i.e., do not lose information with respect to the original property. We show that these widening rules can be used to achieve termination in cases where termination is not guaranteed in forward analysis. We also show that for some examples for which termination of forward analysis but widening can drastically accelerate the termination.

In general, the events considered here may not be an original event but is constructed as a composition of events. We write $e = \mathit{event}(\gamma, \varphi)$ when application of the event e to the constraint γ results in the constraint φ.

Definition 1 (Compound Events.). *Let* $e_1 \equiv$ **cond** ψ_1 **action** $\varphi_1, \ldots, e_k \equiv$ **cond** ψ_k **action** φ_k *be* k *original events of the timed system. Assume that the source location for the first event and the target location for the last event are the same. Assume that the target location for the* j*th event and the source location for the* $(j+1)$*st event are same* $(1 \le j \le k-1)$*. Also assume that for each event* e_j*, each variable* x_i *and* x'_i *in the guard and action have been alpha-renamed to* x_i^j *and* x_i^{j+1} *respectively. Then the compound event (or composed event) corresponding to* $e_1, \ldots, e_k$ *is given by*[4] **cond** *true* **action** $\varphi \wedge \psi$ *where* $\varphi \wedge \psi$ *is given by*

$$\varphi \wedge \psi \equiv (\exists_{-\{\mathbf{x^1}, \mathbf{x^{k+1}}\}} \varphi_1 \wedge \psi_1 \wedge \ldots \wedge \varphi_k \wedge \psi_k)[\mathbf{x}, \mathbf{x'}].$$

See below for examples of compound events. Given that the theory of reals with addition and order admits quantifier elimination, $\varphi \wedge \psi$ can be expressed in a conjunctive normal form. For the timed automaton given in Figure 1, the event **cond** *true* **action** $x' = x + z, z \ge 0, y' = y + z, y' \le 2$ is a simple (time) event corresponding to the time transition in location 0 while **cond** $\mathit{loc} = 0$ **action** $y' = 0, x' = x, \mathit{loc}' = 1$ is a simple or original (edge) event corresponding to the edge from location 0 to 1.

We consider only non-strict inequalities here. The strict inequalities can be dealt with similarly. The template for symbolic boundedness procedure with widening is defined in Figure 7 in the Appendix. The function WIDEN is defined in Figure 8 in the Appendix. Note that the procedure in Figure 8 is based on a breadth-first search. In a call to $\mathit{WIDEN}(\Phi_i, \mathit{post}(\Phi_i))$ one of the three widening rules WIDEN_1, WIDEN_2 or WIDEN_3 described below is fired provided the conditions of that rule are satisfied. If the condition in the WIDEN function applies to several decompositions of γ, the corresponding widenings are effectuated in several successive iterations. In the sequel, we refer to the procedure Symbolic-Boundedness-W as the breadth first forward reachability analysis procedure with widening.

We now illustrate the widening rules with examples. The intuition behind the widening rules is as follows: if we can detect from the syntax of a sequence of events $\bar{e}$ and a constraint φ, that the sequence $\varphi, \mathit{post}_{|\bar{e}}(\varphi), \ldots$ "grows" infinitely in a particular direction (i.e., actually leads to an infinite sequence with respect to reachability analysis), we will try to add the union of the sequence to our set of reachable states. Thus for widening rule I (for the if part), the syntax of the

[4] Note that we can construct an event with empty guard as the events **cond** ψ **action** φ and **cond** *true* **action** $\varphi \wedge \psi$ are equivalent with respect to symbolic model checking.

input constraint ($\eta \wedge x_i - x_j \geq c_{ij}$) and that of the event ($\theta \wedge x'_j = x_j + x'_i \wedge x_i \leq c_i$ which may be a composition of several simple events as described above) tells us that this constraint-event combination will generate an infinite behavior ($\eta \wedge x_i - x_j \geq c_{ij}$, $\eta \wedge x_i - x_j \geq c_{ij} - c_i$, $\ldots$; see example below) provided the other conditions are satisfied. Hence we infer the limit of this sequence which is η (since $c_{ij} \leq 0$ and $c_i > 0$) and add it to the set of states. Similar are the intuitions behind the other widening rules.

Consider the example timed automaton in Figure 2. Note that forward breadth-first reachability analysis does not terminate. Consider the events 4 and 3. Event 4 is given by $e \equiv$ **cond** $x_2 \leq 2$ **action** $x'_2 = 0, x'_1 = x_1$ (we do not show the location explicitly). Event 3 is the time event at location 1 and is given by $e' \equiv$ **cond** *true* **action** $x'_1 = x_1 + z, x'_2 = x_2 + z, z \geq 0$ (time increases by amount z). We compose transition (sometimes we will use the term 'transition' for 'event') 4 and transition 3 using the method given above. The resulting compound event is $e_1 \equiv$ **cond** *true* **action** $\varphi \wedge \psi$ where

$$\varphi \wedge \psi \equiv x'_1 = x_1 + x'_2, x_2 \leq 2, x'_2 \geq 0.$$

Now consider the infinite sequence of states produced by a breadth-first reachability analysis for this automaton

$$\begin{array}{l}
\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle \xrightarrow{1} \langle l\_0(\mathbf{x}), x_1 = x_2, x_1 \geq 0\rangle \\
\xrightarrow{2} \langle l\_1(\mathbf{x}), x_1 = 0, x_2 \geq 0\rangle \xrightarrow{3} \langle l\_1(\mathbf{x}), x_2 - x_1 \geq 0, x_1 \geq 0\rangle \\
\xrightarrow{4} \langle l\_1(\mathbf{x}), 0 \leq x_1 \leq 2, x_2 = 0\rangle \\
\xrightarrow{3} \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0, x_2 - x_1 \geq -2\rangle} \\
\xrightarrow{4} \langle l\_1(\mathbf{x}), 0 \leq x_1 \leq 4, x_2 = 0\rangle \\
\xrightarrow{3} \langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0, x_2 - x_1 \geq -4\rangle \\
\xrightarrow{4} \ldots
\end{array}.$$

(in the above we denote location i by l_i.) Now see that the state under the overbrace along with event e_1 satisfies the conditions of the widening rule I (the *if* part) defined in Figure 10 in the Appendix ($i = 2, j = 1, \gamma \equiv \eta \wedge x_2 - x_1 \geq -2$ where $\eta \equiv x_1 - x_2 \geq 0, x_2 \geq 0$, $c_{21} = -2$ and $\theta \equiv x'_2 \geq 0$). Hence, applying the widening, we obtain the state $\langle l_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$ (the reader can easily make out that if the sequence of transition 4 and transition 3 is repeated infinitely many times to the state under the overbrace, the constraint $x_1 - x_2 \geq 0, x_2 \geq 0$ will be obtained). After this any state generated is subsumed (included) by this state. Hence the breadth first forward reachability analysis with widening terminates.

Before defining widening rule II, let us introduce some notation. Let $\mathcal{N}_n$ denote $\{1, \ldots, n\}$. Let I denote a subset of $\mathcal{N}_n$. The widening rule II is defined in figure 11 in the Appendix.

To show an example in which application of widening rule II forces termination, we look at the example in figure 3. Note that breadth-first forward reachability analysis does not terminate for this example. The following infinite

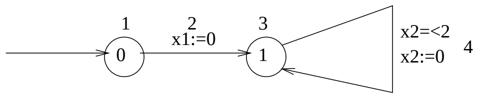


Fig. 2. Illustrating widening rule I

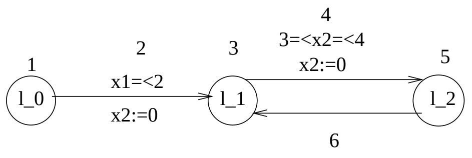


Fig. 3. Illustrating widening rule II

sequence of states is generated in a breadth-first forward reachability analysis for this example.

$$
\begin{aligned}
&\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle \\
&\xrightarrow{1} \langle l\_0(\mathbf{x}), x_1 = x_2, x_2 \geq 0\rangle \\
&\xrightarrow{2} \langle l\_1(\mathbf{x}), x_1 \geq 0, x_1 \leq 2, x_2 = 0\rangle \xrightarrow{3} \langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 - x_1 \geq -2, x_2 \geq 0\rangle \\
&\xrightarrow{4} \langle l\_2(\mathbf{x}), x_1 \geq 3, x_1 \leq 6, x_2 = 0\rangle \xrightarrow{5} \langle l\_2(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle \\
&\xrightarrow{6} \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle} \\
&\xrightarrow{3} \langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle \\
&\xrightarrow{4} \langle l\_2(\mathbf{x}), x_1 \geq 6, x_1 \leq 10, x_2 = 0\rangle \\
&\xrightarrow{5} \langle l\_2(\mathbf{x}), x_1 - x_2 \geq 6, x_2 - x_1 \geq -10, x_2 \geq 0\rangle \\
&\xrightarrow{6} \langle l\_1(\mathbf{x}), x_1 - x_2 \geq 6, x_2 - x_1 \geq -10, x_2 \geq 0\rangle \ldots
\end{aligned}
$$

Now consider the compound event $e_2 \equiv$ **cond** *true* **action** $\varphi \wedge \psi$ obtained by composing transitions 3, 4, 5 and 6. Here

$$\varphi \wedge \psi \equiv x_1' \geq x_1 - x_2 + x_2' + 2 \wedge x_1' - x_2' \leq x_1 - x_2 + 3 \wedge x_1' \geq x_1 + x_2' \wedge x_2' \geq 0.$$

See that the conditions of widening rule II (the *if* part) are satisfied for e_2 and the state under the overbrace in the sequence ($i = 2, j = 1$, $\eta \equiv x_2 \geq 0$, $c_{21} = -6 < 0$, $c_{12} = 3$ and $\theta \equiv x_1' \geq x_1 - x_2 + x_2' + 2, x_1' \geq x_1 + x_2', x_2' \geq 0$). The reader can easily convince herself that the give state and event e_2 do not satisfy the conditions of widening rule I). Applying the widening, we obtain the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 \geq 0\rangle$ (viewing the constraint solving involved geometrically may provide better intuitions). The states which are further generated are subsumed by this state. So breadth-first forward reachability analysis with widening terminates after this. Note that in this case, application of abstract interpretation with the convex hull operator as is done in [24,3,19] would produce the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$. This can lead to 'don't know' answers to certain reachability questions (e.g., consider the reachability question whether the location l_1 can be reached with the values of the clocks satisfying the constraint $x_1 - x_2 > 2, x_2 - x_1 > -3, x_2 \geq 0)$. As for the extrapolation abstraction [12], we have already stated in the Introduction that it is unsuitable for model checking for boundedness properties.

In widening rule III we use periodic sets following Boigelot and Wolper [9].

The widening rule III is defined in Figure 6 in the Appendix, where the predicate $int(x)$ is true if and only if x is a nonnegative integer. Consider the example in Figure 4. Note that breadth-first forward reachability analysis does not terminate for this example. The following infinite sequence of states is generated in course of a forward (breadth-first) reachability analysis for this example:

$$\begin{aligned}
&\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle \\
&\xrightarrow{1} \langle l\_0(\mathbf{x}), x_1 = x_2, x_2 \geq 0\rangle \\
&\xrightarrow{2} \langle l\_1(\mathbf{x}), x_2 = 0, x_1 \geq 0, x_1 \leq 1\rangle \xrightarrow{3} \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 - x_1 \geq -1, x2 \geq 0\rangle} \\
&\xrightarrow{4} \langle l\_2(\mathbf{x}), x_1 \geq 4, x_1 \leq 5, x_2 = 0\rangle \xrightarrow{5} \langle l\_2(\mathbf{x}), x_1 - x_2 \geq 4, x_2 - x_1 \geq -5, x_2 \geq 0\rangle \\
&\xrightarrow{6} \langle l\_1(\mathbf{x}), x_1 - x_2 \geq 4, x_2 - x_1 \geq -5, x_2 \geq 0\rangle \ldots
\end{aligned}$$

Now we compose transitions 4, 5 and 6. The compound event is $e_3 \equiv$ **cond** *true* **action** $\varphi \wedge \psi$ where

$$\varphi \wedge \psi \equiv x_1' = x_1 + x_2' \wedge x_2' \geq 0 \wedge x_2 = 4.$$

It is easy to see that the state under the overbrace in the infinite sequence along with event e_3 satisfies the conditions of widening rule III ($i = 2, j = 1$, $\eta \equiv x_2 \geq 0$, $c_{12} = 0$, $c_{21} = -1 < 0$). Hence, applying widening rule III we get the state $\langle l\_1(\mathbf{x}), \exists k \geq 0, int(k), x_1 - x_2 \geq k * 4, x2 - x_1 \geq -1 - k * 4\rangle$. The states further generated are subsumed by this state. So (breadth-first) forward reachability analysis terminates after applying the widening rule. Note that application of abstract interpretation with the convex hull operator [19,3,24] will produce the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$. Hence for certain reachability questions we can get a 'don't know' answer.

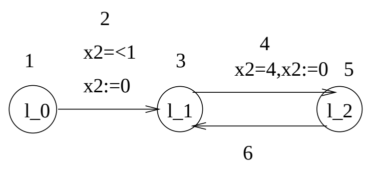


Fig. 4. Illustrating the widening rule III

Now we show that the widening rules are accurate with respect to boundedness properties.

Theorem 1 (Soundness and Completeness). *The procedure Symbolic-Boundedness-W obtained by abstracting the forward breadth first reachability* *analysis procedure with widening defined by the widening rules I, II and III yields (if terminating) a full test of boundedness (unboundedness) properties for timed systems (modeled by timed automata).*

Note that the above theorem also implies that if the procedure Symbolic-Boundedness-W terminates, then one can get a full test of safety properties as well. Below we provide effective sufficient conditions for termination of Symbolic-Boundedness-W. By a *simple path* in a timed automaton $\mathcal{U}$, we mean a sequence of events $e_1 \ldots e_m$ where each e_i is an original event of $\mathcal{U}$ and

- the source location of e_{i+1} is the same as the target location of e_i for $1 \leq i \leq m-1$,
- any event e_i with same source and target locations is a time event,
- for any two edge events e_i and e_j, $1 \leq i < j < m$, the target locations of e_i and e_j are different,
- and if e_i is an (original) time event, then e_{i-1} and e_{i+1} are edge events.

With this definition, there are only a finite number of such simple paths in a timed automaton. The simple path $p = e_1 \ldots e_m$ leads from location ℓ^1 to the location ℓ^2 if there is a the source location of e_1 is ℓ^1 and the target location of e_m is ℓ^2. The simple path $e_1 \ldots e_m$ is a simple cycle if the source location of e_1 is the same as the target location of e_m. Note that there is only a finite number of such simple cycles in a timed automaton.

Theorem 2 (Sufficient Conditions for Termination). *Let $\mathcal{U}$ be a timed automaton and let ℓ be a location in $\mathcal{U}$ such that there is a simple cycle C from ℓ to itself and the following three conditions are satisfied.*

- *There is a simple path in $\mathcal{U}$ of the form $e \equiv$ **cond** φ **action** ψ leading from the initial location ℓ^0 to ℓ such with the cycle C along with the constraint $(\exists_{-\mathbf{x}'}\varphi^0 \wedge \varphi \wedge \psi)[\mathbf{x}]$ that satisfies the conditions of the widening rules I, II or III where φ^0 is the initial constraint.*
- *For each original event $e' \equiv$ **cond** φ' **action** ψ' with target location ℓ that lies on a cycle in the control graph of $\mathcal{U}$, $(\exists_{-\mathbf{x}'}\varphi' \wedge \psi')[\mathbf{x}] \models post_{\mid t}(\eta)$ if widening rule I or II is satisfied in the previous condition and $(\exists_{-\mathbf{x}'}\varphi' \wedge \psi')[\mathbf{x}] \models post_{\mid t}(\eta \wedge \exists k \geq 0 \wedge int(k) \wedge x_i - x_j \geq c_{ji} + k * c_j \wedge x_j - x_i \geq c_{ji} - k * c_j)$ if widening rule III is satisfied in the previous condition, where η, c_{ji} are as in the definition of the widening rules and t is the time event at location ℓ.*
- *The control graph of $\mathcal{U}$ satisfies the temporal formula $AG(true \implies AF(at\_\ell))$ where at_ℓ is an atomic proposition satisfied only by location ℓ.*

Then the procedure Symbolic-Boundedness-W terminates for $\mathcal{U}$.

It can be seen that the example in Figure 2 satisfies the sufficient conditions stated above.

We have implemented a prototype based on the approach (in the CLP($\mathcal{R}$) system of Sicstus Prolog 3.7). The performance shown, so far, by our approach has been quite encouraging. We have used our implementation to verify the safety and boundedness properties of several well-known benchmark examples taken from literature. The experimental results are summarized in the table in Figure 5. All results are obtained on a PC (200 MHz Pentium Pro). The experiments show a marked improvement over the timings obtained without using the accurate widening rules in [23]. The timings obtained for Fischer's protocol (two processes), Rail-Road Crossing, and Audio Protocol without using the widening rules are $4.2s$, $1.8s$ and $7.2s$ respectively. All the timings in Figure 5 denote the total time taken for reachability analysis.

Example	*time (seconds)*
Fischer's Protocol (Two Processes) [22]	2.1
Rail-road Crossing	0.8
Audio Protocol [20]	2.3

Fig. 5. Experimental results

4 Related Work

In this paper, we have presented a constraint based framework for symbolic model checking of timed systems against boundedness properties. We have shown that it is possible to achieve (or just accelerate) termination of our symbolic model checking procedure with abstractions by widening that are, as we prove, accurate. Our approach allows us to do a full test of the safety and boundedness (unboundedness) properties without going into the complications of region construction. Regarding the generality of our approach, we do not claim that the three widening rules described in this paper encompass (i.e., achieves termination and/or speed-up of model checking procedure) the full class of timed automata. We have provided sufficient conditions under which the procedure Symbolic-Unboundedness-W is guaranteed to terminate. However, for several examples the procedure terminates even though the sufficient conditions do not apply.

Note that the model checking procedure for Uppaal [8] is also based on semantics of constraints but their algorithms are based on graph-theoretic techniques rather than techniques from constraint programming. We believe that incorporation of accurate widening framework in UPPAAL and the other approaches mentioned above can significantly speed-up model checking procedures based on those approaches.

Our widening operatoris closely related to Boigelot and Wolper's loop-first technique [9] for deriving periodic sets as representations of infinite sets of integer valued states for reachability analysis. As a difference, Boigelot and Wolper analyze cycles and nested cycles in the control graph to detect meta-transitions *before* and independently of their forward model checking procedure, whereas we construct new events *during* our model checking procedure and consider them only if we detect that they possibly lead to an infinite loop. Berard' and Fribourg [5] use a constraint-based framework for reachability analysis for timed Petri nets. They have been able to verify several interesting examples using their approach based on meta transitions. Our approach, rooted in the abstract interpretation framework, is different from theirs in that we accelerate the model checking procedure using widening rules based on syntax.

The application of widening techniques to the verification of systems with huge or infinite state spaces has proven useful in several examples. Halbwachs et.al. [19], using linear relational analysis to prove properties for *linear hybrid systems*, defines a widening operator over convex polyhedra: unions of convex polyhedra are approximated by their convex hull before the widening step. Approximation techniques for more general classes of hybrid systems are studied in [15,14]. Specifically, Henzinger and Ho [14] apply an extrapolation operator which gives better approximations than Halbwachs et. als' convex widening operator in their examples. For integer valued systems, abstract interpretation has been used effectively in [6]. In [7], it was explicitly mentioned that one main difficulty with the approximate approach is that the abstraction is often too rough. We have shown in Section 3 that our widening techniques will give full test of reachability properties for timed systems where the approximate methods [3, 24,19] would produce a 'don't know' answer. Also, in contrast with our accurate widenings, the widening techniques proposed in [3,24,19] cannot be used for model checking for boundedness properties. Note that it is not possible to find out in most cases, using semantics-based techniques, whether a program loop really generates an infinite behavior with respect to reachability analysis. Hence, application of widening combined with semantics-based techniques may result in loss of accuracy that will render these techniques unsuitable for model checking for boundedness properties. It would be interesting to look at how the techniques described in this paper extend to more general classes of hybrid systems. The general goal will be a whole library of accurate widening rules for a variety of verification problems.

References

1. R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. Minimization of timed transition systems. In R. Cleaveland, editor, *CONCUR: Concurrency Theory*, volume 630 of *LNCS*, pages 340–354. Springer-Verlag, 1992.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.
3. F. Balarin. Approximate reachability analysis of timed automata. In *17th IEEE Real-Time Systems Symposium*, pages 52–61. IEEE Computer Society Press, 1996.
4. B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In O. Grumberg, editor, *CAV'97: Computer Aided Verification*, volume 1254 of *LNCS*, pages 167–178. Springer-Verlag, 1997.
5. B. Berard' and L. Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In J. C. M. Baeten and S. Mauw, editors, *CONCUR: Concurrency Theory*, volume 1664 of *LNCS*, pages 178–193. Springer-Verlag, 1999.

6. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetics. In Orna Grumberg, editor, *the 9th International Conference on Computer Aided Verification (CAV'97)*, LNCS 1254, pages 400–411. Springer, Haifa, Israel, July 1997.
7. T. Bultan, R. Gerber, and W. Pugh. Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results, february 1998.
8. Johan Bengtsson, Kim. G. Larsen, Fredrik Larsson, Paul Petersson, and Wang Yi. Uppaal in 1995. In T. Margaria and B. Steffen, editors, *TACAS*, LNCS 1055, pages 431–434. Springer-Verlag, 1996.
9. Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In David Dill, editor, *6th International Conference on Computer-Aided Verification*, volume 818 of *LNCS*, pages 55–67. Springer-Verlag, June 1994.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th ACM Symposium on Principles of Programming Languages*, 1977.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1978.
12. C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In Bernhard Steffen, editor, *TACAS98: Tools and Algorithms for the Construction of Systems*, LNCS 1384, pages 313–329. Springer-Verlag, March/April 1998.
13. N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *the International Conference on Computer-Aided-Verification*, volume 697 of *LNCS*, pages 333–346. Springer-Verlag, 1993.
14. T. A. Henzinger and P.-H. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, LNCS 999, pages 252–264. Springer-Verlag, 1995.
15. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. In O. Grumberg, editor, *CAV97: Computer-aided Verification*, LNCS 1254, pages 460–463. Springer-Verlag, 1997.
16. Thomas. A. Henzinger and Orna Kupferman. From quantity to quality. In Oded Maler, editor, *Hybrid and Real-Time Systems International Workshop,Hart '97*, volume 1201 of *LNCS*, pages 48–62, Grenoble, France, March 1997. Springer-Verlag.
17. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.
18. T. A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV'98: Computer-aided Verification*, LNCS 1427, pages 195–206. Springer-Verlag, 1998.
19. N. Halbwachs, Y-E. Proy, and P. Romanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
20. Pei-Hsin Ho and Howard Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *the Seventh Conference on Computer-Aided Verification*, pages 381–394, Liege, Belgium, 1995. Springer-Verlag. LNCS 939.
21. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

22. K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model checking of real-time systems. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.
23. S. Mukhopadhyay and A. Podelski. Model checking for timed logic processes. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL: Computational Logic*, LNCS, pages 598–612. Springer, 2000. Available at http://www.mpi-sb.mpg.de/∼supratik/.
24. H. Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Stanford University, 1995.

A Algorithms

```
Function $WIDEN_3(\gamma, \varphi')$
$if \begin{cases} \gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji} \\ \varphi \wedge \psi \equiv \theta \wedge x'_i = x_i + x'_j \wedge x_j = c_j \\ c_{ij} \leq 0 \\ c_i > 0 \\ c_{ji} \geq c_{ij} \\ \eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji} \wedge x_j = c_j) \end{cases}$
return   $\eta \wedge \exists k \geq 0 \wedge int(k) \wedge x_i - x_j \geq c_{ji} + k * c_j \wedge x_j - x_i \geq c_{ji} - k * c_j$
else   return   $\varphi'$
```

Fig. 6. Widening Rule III

```
Procedure Symbolic-Boundedness-W($\Phi$)
Input A set of constraints $\Phi$
Output A set of constraints representing sets of states reachable from [$\Phi$]
$\Phi_0 := \Phi$.
repeat
begin
$\Phi_{i+1} = \Phi_i \cup WIDEN(\Phi_i, post(\Phi_i))$
end
until $\Phi_{i+1} \models \Phi_i$.
return $\Phi_i$.
```

Fig. 7. Template for Model Checking for Boundedness Properties with Widening

```
**Function** $WIDEN(\Gamma, \Phi) = \{WIDEN(\gamma, \varphi) \mid \gamma \in \Gamma, \varphi \in \Phi\}$
**Function** $WIDEN(\gamma, \varphi)$
$\varphi_1 := WIDEN_1(\gamma, \varphi)$
**If** $\varphi_1 \not\equiv \varphi$ **return** $\varphi_1$
**else** $\{\varphi_1 := WIDEN_2(\gamma, \varphi)$
**If** $\varphi_1 \not\equiv \varphi$ **return** $\varphi_1$
**else** $\varphi_1 := WIDEN_3(\gamma, \varphi)\}$
**return** $\varphi_1$
```

Fig. 8. Widen Function

```
**Procedure** Symbolic-Boundedness($\Phi$)
**Input** A set of constraints $\Phi$
**Output** A set of constraints representing sets of states reachable from $[\Phi]$
$\Phi_0 := \Phi$.
**repeat**
**begin**
$\Phi_{i+1} = \Phi_i \cup post(\Phi_i)$
**end**
**until** $\Phi_{i+1} \models \Phi_i$.
return $\Phi_i$.
```

Fig. 9. Template for Model Checking for Boundedness Properties

```
Function $WIDEN_1(\gamma, \varphi')$
if $\begin{cases} \gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \\ \varphi \wedge \psi \equiv \theta \wedge x'_j = x_j + x'_i \wedge x_i \leq c_i \\ c_{ij} \leq 0 \\ c_i > 0 \\ \eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i) \end{cases}$
or if $\begin{cases} \gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i \\ \varphi \wedge \psi \equiv \theta \wedge x'_j = x_j + x'_i \\ c_{ij} \leq 0 \\ c_i > 0 \\ \eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i) \end{cases}$
return   $\eta$
else   return   $\varphi'$
```

Fig. 10. Widening Rule I

```
Function $WIDEN_2(\gamma, \varphi')$
if $\begin{cases} \gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji} \\ \varphi \wedge \psi \equiv \theta \wedge x'_j - x'_i \leq x_j - x_i + a_{ji} \\ c_{ij} \leq 0 \\ a_{ji} > 0 \\ (\exists_{-\mathbf{x}'}\varphi \wedge \psi \wedge \eta) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji}) = \zeta \wedge x'_j - x'_i \geq c'_{ji} \\ \eta[\mathbf{x}'] \models \zeta \\ 0 \leq c'_{ji} \leq -c_{ij} \end{cases}$
return   $\eta \wedge x_j - x_i \geq c_{ji}$
else   if $\begin{cases} \gamma \equiv \eta \wedge \bigwedge_{i,j,\in I} x_i - x_j \geq c_{ij} \\ \varphi \wedge \psi \equiv \theta \wedge \bigwedge_{i,j,\in I} x_j - x_i \leq x_j - x_i + a_{ji} \\ c_{ij} \leq 0 \\ a_{ji} > 0 \\ \eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}\varphi \wedge \psi \wedge \eta) \wedge (\exists_{-\mathbf{x}'}\theta \wedge \bigwedge_{i,j,\in I} x_i - x_j \geq c_{ij}) \end{cases}$
return   $\eta$.
else   return   $\varphi'$
```

Fig. 11. Widening Rule II

Adaptive Saturation-Based Reasoning

Alexandre Riazanov and Andrei Voronkov

University of Manchester
{riazanov,voronkov}@cs.man.ac.uk

Abstract. For most applications of first-order theorem provers a proof should be found within a fixed time limit. When the time limit is set, systems can perform much better by using algorithms other than the ordinary complete ones. In this paper we describe the Limited Resource Strategy intended to improve performance of resolution- and paramodulation-based provers when a fixed limit is imposed on the time of a run. We give experimental evidence that the Limited Resource Strategy gives a significant improvement over the OTTER saturation algorithm, algorithms not using passive clauses for simplification and the weight-based algorithms.

1 Reasoning with Limited Resources

First-order theorem provers (in the sequel simply *provers*) have a number of applications. The main application areas for provers are software and hardware verification, computer algebra and assisting human mathematicians. In nearly all applications, provers are used in the following way. When a prover is run on a goal, a time limit is set. If neither proof nor countermodel could be found within the time limit, the prover terminates. Then the goal can be reconsidered, for example by formulating intermediate statements (lemmas) or by providing some inference steps interactively, and the proof-search continues on the new goals or using the lemmas.

Since first-order logic is undecidable, any complete prover is potentially non-terminating. Therefore setting a time limit for processing a particular goal is a natural idea. Moreover, for most applications it is difficult to expect human users or systems ready to wait for an answer forever. This paper addresses the problem of reasoning in limited time. It turns out that when the time is limited, systems can perform much better by using algorithms other than ordinary complete ones. In this paper we describe the so-called Limited Resource Strategy (LRS) implemented in our system Vampire [10,9], discuss its advantages and drawbacks, compare it with the strategies so far used in Vampire and other systems and describe experiments carried out over a number of first-order problems.

2 Saturation-Based Theorem Proving

The fastest first-order theorem provers of the last two CASC competitions [13] (with one exception of Setheo [7]) use saturation algorithms. There exist two main kinds of saturation algorithms, one was implemented in OTTER [6] and its predecessors (an overview of these early provers can be found in [5]), another one was used for the first time in DISCOUNT [1]. In this section we describe these saturation algorithms. For simplicity, we will call them the OTTER and the DISCOUNT algorithms, respectively. The OTTER algorithm is implemented at least in OTTER, Gandalf [14], SPASS [16], and Vampire, and the DISCOUNT algorithm is implemented at least in DISCOUNT, E [11], SPASS, Vampire and Waldmeister [4].

Unfortunately, the terminology related to saturation algorithms used in different research groups varies a lot, so no standard terminology exists. Let us develop some relevant terminology. Saturation algorithms used in first-order theorem provers operate on *clauses*. For each new clause generated by an inference the prover decides whether this clause should be *kept* or discarded. The set of kept clauses may be huge, so most of the systems perform inferences not on *all* kept clauses, but only on a subset of them. The clauses in this subset, i.e. those used for inferences will be called *active*. All other kept clauses are *passive* (though as we will see passive clauses can also participate in inferences, but of a special kind).

The two different saturation algorithms differ in their treatment of passive clauses. In the DISCOUNT algorithm passive clauses never participate in inferences or simplifications. In the OTTER algorithm passive clauses can participate in simplifications, for example rewriting by unit equalities or subsumption.

2.1 The OTTER Saturation Algorithm

Let us begin with the OTTER algorithm which is shown in Figure 1. It is parametrized by several procedures explained below:

- *select* is the clause selection function. It decides which passive clause should be selected for activation.
- *infer* is the function that performs inferences between the current clause *current* and the set of active clauses *active*. This function returns the set of clauses obtained by all such possible inferences. This function varies from system to system. Usually, *infer* applies inferences in some complete inference system of resolution with paramodulation.
- *simplify*(set, by) is a procedure that performs simplification. It deletes redundant clauses from *set* and simplifies some clauses in *set* using the clauses in *by*. To preserve completeness, the simplified clauses are always moved to *passive*. Typically, deleted clauses include tautologies and those clauses subsumed by clauses in *by*. A typical example of simplification is rewriting by unit equalities in *by*.
- Likewise, *inner_simplify* simplifies clauses in *new* using other clauses in *new*.

When we simplify *new* using the clauses in $active \cup passive$, we speak of *forward simplification*, when we simplify *active* and *passive* using the clauses in *new*, we speak of *backward simplification*.

Typical behaviour of this algorithm is quantitatively characterised by the following empirical observation: in a matter of seconds the total number of kept

```
input: $init$ : set of clauses ;
var $active$, $passive$, $new$ : sets of clauses ;
var $current$ : clause ;
$active$ := $\emptyset$ ;
$passive$ := $init$ ;
while $passive \neq \emptyset$ do
  $current$ := $select(passive)$ ;
  $passive$ := $passive - \{current\}$ ;
  $active$ := $active \cup \{current\}$ ;
  $new$ := $infer(current, active)$ ;
  if $goal\_found(new)$ then return $provable$ ;
  $inner\_simplify(new)$ ;
  $simplify(new, active \cup passive)$ ;
  if $goal\_found(new)$ then return $provable$ ;
  $simplify(active, new)$ ;
  $simplify(passive, new)$ ;
  if $goal\_found(active \cup passive)$ then return $provable$ ;
  $passive$ := $passive \cup new$
od ;
return $unprovable$
```

Fig. 1. The OTTER Saturation Algorithm

clauses gets very big, whereas the share of the active clauses is small and keeps decreasing. To illustrate this, we provide statistics on an unsuccessful run of Vampire with the time limit of 1 minute on the TPTP problem ANA003-1. During this run, 261,573 clauses were generated. The overall number of active clauses was 1,967, the overall number of passive clauses 236,389. The clauses generated in this run contain function symbols and equality, many of the clauses have a large number of literals and/or heavy terms. It was possible to process such a large number of clauses in one minute due to success of the modern indexing techniques [3,8]. Even when the state-of-the-art indexing techniques are used, it is very difficult to manage clause sets containing over $100,000$ clauses efficiently. As a consequence, when theorem provers are used for practical applications, completeness is often compromised in favour of efficiency: the provers discard clauses that may be nonredundant.

2.2 The DISCOUNT Saturation Algorithm

It was observed that usually the total number of active clauses in the OTTER algorithm is considerably less than the number of passive clauses. Meanwhile, simplification of and by the passive clauses consumes significant amount of time. One can modify the OTTER saturation algorithm in such a way that passive clauses never participate in simplifications. Such a modified saturation algorithm will be called *the DISCOUNT algorithm* in this paper. The algorithm is shown in Figure 2.

```
input: $init$ : set of clauses ;
var $active, passive, new$ : sets of clauses ;
var $current$ : clause ;
$active$ := $\emptyset$ ;
$passive$ := $init$ ;
while $passive \neq \emptyset$ do
  $current$ := $select(passive)$ ;
  $passive$ := $passive - \{current\}$ ;
  $simplify(\{current\}, active)$ ;
  if $current$ is simplified to a goal return $provable$
  if $current$ is not simplified to a tautology
  then do
    $simplify(active, \{current\})$ ;
    if $goal\_found(active)$ then return $provable$ ;
    $active$ := $active \cup \{current\}$ ;
    $new$ := $infer(current, active)$ ;
    if $goal\_found(new)$ then return $provable$ ;
    $simplify(new, active)$ ;
    $passive$ := $passive \cup new$
  od ;
od ;
return $unprovable$
```

Fig. 2. The DISCOUNT Saturation Algorithm

Compared to the OTTER saturation algorithm, this algorithm has the following features:

– The new clauses are forward simplified by the active clauses only, the passive clauses don not take part in this.
– Neither active nor passive clauses are backward simplified by the retained new clauses.
– After selection of the current clause it is simplified again by the active clauses and then is itself used to simplify the active clauses. Again, the passive clauses are not affected.

If we assume that the overall number of kept clauses is significantly larger than the number of used ones, this algorithm involves less computation for the same number of active clauses than the OTTER algorithm. However, this algorithm has a severe weakness: It performs a very limited amount of backward simplification steps compared to the OTTER algorithm. This sometimes results in the following effect: finding some proofs that are quickly found by the OTTER algorithm involving backward simplification, is now delayed significantly. To illustrate this, consider a simple example. Suppose that a, b are constants and generated are two unit clauses $t = a$ and $t = b$ with a heavy term t. Since most provers try to select lighter clauses (with fewer function symbols), it may take a long time before these clauses both appear among the active ones. Rewriting the latter clause by the former one gives a very light clause $a = b$ which is very likely to contribute to a derivation of the empty clause. This simplification from $t = a$ into $t = b$ will be discovered by the OTTER algorithm as soon as both $t = a$ and $t = b$ have been generated, but it may take a long time before they get selected by the DISCOUNT algorithm. The same applies for other selection criteria, for example when older clauses are preferred to younger ones, since $t = a$ and $t = b$ can as well be younger clauses, so their selection will be delayed.

It was observed experimentally that the time spent for storing and retrieving passive clauses in the DISCOUNT algorithm is negligible compared to the overall runtime. Therefore, one cannot expect to improve considerably the performance of the DISCOUNT algorithm by, e.g. trying to discard some passive clauses when a time limit is set (though it can save a lot of memory).

3 Reasoning in Limited Time by the OTTER Algorithm

Growth of the number of kept clauses in the OTTER algorithm causes fast deterioration of the rate of processing of active clauses. Thus, when a complete procedure based on the OTTER algorithm is used, even passive clauses with high selection priority often have to wait very long before they contribute to the search. In the provers based on the OTTER algorithm, all solutions to the completeness-versus-efficiency problem are based on the same idea: some nonredundant clauses are discarded from the clause sets *active*, *passive*, or *new*.

In this section we explain several approaches to discarding clauses implemented in the state-of-the-art provers and analyse their main advantages and disadvantages.

3.1 Weight Limit Strategy

The Weight Limit Strategy was implemented already in the very first versions of OTTER. The idea is to set a limit W on the weight of clauses. The weight of a clause is a measure reflecting its complexity, for example the number of symbols in it. All new clauses with the weight greater than W are discarded. This *Weight Limit Strategy* is especially helpful for interactive use and solving difficult problems. The user specifies an arbitrary weight limit, runs the prover on a goal, and studies the output. If a proof was not found, the weight limit can be changed. Weight Limit Strategy is not very useful for fully automatic theorem proving. Since problems submitted to provers are of very diverse nature, the useful weight limits can vary. When the weight limit is too high, there is essentially no difference between a complete algorithm and a weight-limit based one. When the weight limit is too small, a proof with the given weight limit may not exist at all, even when a complete algorithm will easily find a proof. Another problem with the weight limit was observed by the authors in case studies: for many problems heavy clauses are needed for a very short time in the beginning of the proof-search, and then only very light clauses suffice for finding a proof.

3.2 Incremental Weight Limit Strategy

However, since light clauses experimentally proved to be very useful, the Weight Limit Strategy is very appealing. Several provers, including Gandalf [14], Bliksem [2] and Fiesta adopted the *Incremental Weight Limit Strategy.* The idea of this strategy is that the weight limit is initially set to a small value. If no proof is found with this small value, the weight limit is increased, and the proof search begins either from scratch or using the short clauses obtained during the previous run. This strategy is very efficient on quite a number of problems, but also very inefficient on many problems. There are two kinds of problems on which the strategy behaves poorly.

1. Suppose that W is the minimal weight sufficient to find a proof of a problem by the prover's inference system. If the proof search for the weight $W-1$ is too costly, the prover will not increment the weight to W, spending all its time on smaller weights.
2. When heavy clauses are only needed in the beginning of proof-search, the strategy is inefficient, since it will generate and store heavy clauses when it is no more necessary.

3.3 Memory Limit Strategy

This strategy was implemented for the first time in OTTER. The idea is as follows. Some memory limit is set in advance. When $\frac{1}{3}$ of the available memory has been filled, OTTER assigns new weight limit which is calculated in such a way that 5% of passive clauses have smaller weight than the limit. From then on, this recalculation of weight limit is performed after processing every 10 selected clauses.

This strategy has its own disadvantages. The main problem is that the use of memory and the time are loosely connected. If a complete algorithm is run by Vampire, the memory used in 1 minute can vary as much as between 20 and 400 megabytes on a computer with the 400MHz Pentium processor. Setting too low memory limit will make a prover terminate before the time limit because all clauses needed for finding a proof have been discarded. This effect was indeed observed in some previous versions of OTTER intended for the CASC competitions. For example, in the experiments presented in [15] OTTER terminates before the time limit without finding a proof on a number of problems, in the worst case in less than 2 minutes when the time limit was 30 minutes. Setting too high a limit results in considerable slowdown, since then the system behaves as poorly as based on a complete algorithm.

4 Limited Resource Strategy

The main idea of the Limited Resource Strategy is the following. The prover tries to identify which clauses in *passive* and *new* have no chance to be processed by the time limit at all, and discards these clauses. Such clauses will be called *unreachable.* Note that the notion of unreachable clauses is fundamentally different from the notions of redundant ones: redundant clauses are those that can be discarded without compromising completeness at all, the notion of unreachable clauses makes sense only in the context of reasoning with limited resources. How can one identify unreachable clauses?

When the system starts a proof search on a problem, it is also given a time limit t as an argument. The system keeps track of statistics on the use of resources during the proof search. The main resource measure is the average time spent by processing each clause selected as *current*. Note that usually this time increases because the sets *active* and *passive* are growing (*new* is usually relatively small), so the operations *infer* and *simplify* take more and more time. From time to time the system tries to estimate how the proof search statistics would develop towards the time limit and, based on this estimation, identify unreachable clauses.

The main requirements we imposed on the implementation are the following.

Requirement 1. Vampire with LRS should be at least as fast as Vampire using the complete algorithm.

Requirement 2. A proof should not be lost when the time limit is set to an acceptable value. This means roughly the following. Suppose that Vampire with a time limit t_1 has found a proof in time $t_2 < t_1$. Then Vampire with the time limit t_2 should find a proof as well. In other terms, when the time limit is set to t_2 no reachable clause should be lost.

These requirements can be easily satisfied when no clause is identified as unreachable, i.e. by using a complete algorithm. So we have a third requirement

Requirement 3. As many unreachable clauses as possible should be identified as unreachable by Vampire.

Requirement 3 is in conflict with Requirement 2, because the exact estimation of which clauses are unreachable is essentially impossible.

To give the reader an idea how an estimation of unreachable clauses can be done, we give a simple example. Suppose that p clauses have been processed as *current* in t seconds, i.e. p/t clauses per second, and l is the current time limit. If we assume that the proof search will develop at the same pace, in total $p \cdot l/t$ clauses will be processed by the end of the time limit. So if the number of currently kept clauses k is greater than $p \cdot l/t$, then $k - p \cdot l/t$ clauses can be discarded. Of course this estimation may be inaccurate, because the time for processing one clause as *current* will most likely increase. To avoid too big errors, the estimation of p/t must be done frequently enough. In our experiments the estimation was performed after every 500 inferences produced.

The next question is which $k - p \cdot l/t$ clauses should be discarded. The answer can be obtained by applying Requirement 2. One of the consequences of this principle is that no clause processed by the time limit by the complete strategy should be discarded. But the clause selection in the complete algorithm is controlled by the function *select*, so to identify potentially unreachable clauses let us look deeper at the clause selection function.

All modern theorem provers maintain one or more priority queues. Clauses are picked from the priority queues using some *ratios*. By far the most popular design is based on two priority queues: the *age priority queue* gives higher priority to older clauses, the *weight priority queue* to lighter clauses. The rational behind this strategy is based on the following observation: light clauses are easy to process and most likely to contribute to a derivation of the empty clause, but discarding an old clause is more likely to turn an unsatisfiable set of clauses into a satisfiable one than for a younger clause. The system uses a ratio to decide how often the first clause in each queue should be selected. This ratio is called the *pick-given* ratio in OTTER's manual [6], we will call it the *age-weight ratio*. For example, if the age-weight ratio is 1:4, then out of each 5 selected clauses 1 will be taken from the age priority queue and 4 from the weight priority queue. This strategy was introduced for the first time in OTTER and then used by a number of systems, including at least OTTER, Vampire, and Waldmeister [4].

Assume that our clause selection function is based on the age-weight queue design with age-weight ratio $a : w$, which means that out of any $a + w$ clauses a will be selected from the age queue and w from the weight queue. We have decided that $p \cdot (l/t - 1) = p \cdot (l - t)/t$ currently passive clauses can still be processed within the time limit. Of these clauses $a \cdot p \cdot (l - t)/(t \cdot (a + w))$ will be selected from the age priority queue and $w \cdot p \cdot (l - t)/(t \cdot (a + w))$ from the weight priority queue. So Vampire implements a deletion algorithm that discards clauses according to these formulas.

This example shows that LRS can delete many unreachable clauses from *passive*, but it does not demonstrate the full power of the strategy. We will now show that the strategy can have a drastic influence on the way the sets *new* and *active* are processed. This effect is due to the following observation. Suppose that the strategy discarded some clauses, and the maximal weight of the remaining clauses is W. Suppose a new clause C obtained by an inference has a weight $W' \geq W$. We claim this clause is unreachable. Indeed, C cannot be inserted in the reachable part of the weight priority queue. In addition, C is younger than any clause in *passive*. Thus, it cannot be in the reachable part of the age priority queue. So C is unreachable. This means that any future clause with the weight $\geq W$ can be discarded, so we can set the limit on the weight to be $W - 1$. (Note that this limit can go down as soon as the inference process slows down or more clauses with weights less than W are kept).

Apart from using the dynamically changing weight limit W for discarding new clauses, we can also use the weight limit to discard some *kept* clauses. In order to discard a kept clause we have to be sure that any inference with this clause as a parent gives a clause with a weight exceeding W. Resolution-based provers use calculi based on ordered resolution with negative selection. A typical inference rule in such a calculus is ordered resolution with negative selection:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta} \ ,$$

where θ is a most general unifier of A and B, the atom $A\theta$ is maximal in the clause $(A \vee C)\theta$ and $\neg B$ is a literal selected in the clause $\neg B \vee D$. The clause $(C \vee D)\theta$ is called a *resolvent* of the clauses $A \vee C$ and $\neg B \vee D$.

When this rule is applied, the nonmaximal part of $A \vee C$ will always be part of the clause $C \vee D$. Likewise, the non-selected part of $\neg B \vee D$ will be part of

$C \vee D$. It follows that $C \vee D$ is always at least as heavy as the nonmaximal part of $A \vee C$ or the nonselected part of $\neg B \vee D$. The application of the substitution θ to $C \vee D$ yields a clause at least as heavy as $C \vee D$ (unless we factor equal literals). Suppose now that we perform a resolution inference with the clause $A \vee C$. If the weight of C is greater than the weight limit, then any clause inferred from $A \vee C$ would be too heavy. Therefore, $A \vee C$ can be discarded from the search space because it cannot produce a reachable clause. To implement this, when LRS reduces the weight limit, we can search through the whole set *passive* $\cup$ *active* for clauses whose nonmaximal (nonselected) part has a weight greater than W and discard them.

When the weight of C does not exceed W we can sometimes simply compute the weight of $C\theta$. For example, if we have found the substitution θ together with a sufficiently big set of clauses containing the literal $\neg A\theta$, it still might be useful to compute the weight of $C\theta$ and compare it with W in attempt to avoid building all the inferences. Moreover, to estimate weight of $C\theta$ it is often sufficient to have θ constructed only partially. This can be done, for example, when retrieval of literals unifiable with $\neg A$ is being performed in an index, in which case we can identify a branch in the index that does not have to be inspected.

5 Comparison of LRS with Other Approaches

The main feature of LRS over other algorithms is the possibility to adapt to a particular problem based on the runtime information about the proof-search process. No previous knowledge about the problem is needed. In this section we briefly explain some advantages of LRS as compared to other existing approaches.

Weight-limit based approaches. Setting a particular weight limit in the beginning of proof-search can hardly be helpful since the weight limit needed to solve an unknown problem can not be calculated a priori. So we only compare LRS with the Incremental Weight Limit Strategy. This strategy has several well-known pitfalls. Suppose that the strategy is applied to a problem for which the minimal weight limit sufficient to solve the problem is W.

– For some problems, the proof-search with weight limits smaller than W can consume more time than the time limit, while setting the limit to W would solve the problem almost immediately. For such problems the Incremental Weight Limit Strategy is likely to be much less efficient than the complete strategy.
– For some problems, clauses with the weight W are only needed very early in the proof-search, and then clauses with weights less than or equal to some $W' < W$ will suffice. If too many clauses with the weights between W' and W are generated, the strategy can spend too much time on processing these clauses and will fail to find a proof.

The Limited Resource strategy is immune to both kinds of problems. For the first kind of problems, LRS will behave like a complete algorithm, since early in the proof-search LRS behaves like a complete strategy. For the second kind of problems, when LRS discovers that clauses of the weights between W' and W are unreachable, it will discard these clauses.

The DISCOUNT algorithm. The DISCOUNT algorithm behaves poorly for problems which require many backward simplification steps. Backward simplification steps are performed only when the simplifying clause becomes active. As a consequence, sometimes the algorithm cannot find proofs easily found by other strategies, especially when the proofs contain simplification steps between heavy clauses.

Shortcomings of LRS. Ideally, the requirements for LRS guarantee that it should not lose proofs found by a complete strategy. In reality, mistakes in calculating unreachable clauses are unavoidable, so in practice on some problems the complete algorithm beats LRS.

The main reason for miscalculating reachability of clauses is backward simplifications. When an LRS-based algorithm discards clauses beyond the dynamically set weight limit, it is possible that a simplification of a discarded clause would result in a short proof.

However, our experiments carried out over a large number of problems demonstrate that on the average the performance of the LRS-based algorithm is superior to other algorithms.

6 Experimental Comparison of the DISCOUNT Algorithm and the LRS-Based Algorithm

To compare the LRS-based algorithm with the DISCOUNT algorithm, we implemented the DISCOUNT algorithm in Vampire and made a number of experiments on two benchmarks suites: (i) all 3340 clausal problems in TPTP, (ii) the 1836 problems from the list software reuse application (see [12]). On each problem, Vampire was run with 3 different literal selection functions using the DISCOUNT algorithm, and with the same 3 different selection functions using the LRS-based algorithm. The time limit for each run was set to 60 seconds. We used a computer with the 450 MHz Pentium processor. Therefore, altogether we compared the two algorithms on 15,528 tests. The comparison is fair for the following reason. Both algorithm used the same computer, algorithms, datastructures, and memory management. The only difference was on the top-level loop.

To summarise the results, we consider only so-called *interesting* tests. Intuitively, a test is interesting if it is neither too easy nor too difficult. Formally, we consider the following criterion. A test is interesting if one of the following conditions is satisfied:

1. exactly one of the two algorithms solved the problem; or
2. both algorithms solved the problem, and at least one of them spent more than 10 seconds on it.

The results are presented in Figure 3. In this figure we use the following notation:

++: the problem was solved only by the LRS-based algorithm;
+: the problem was solved by both algorithms, but the LRS-based one was at least 10 seconds faster;
=: the problem was solved by both algorithms, and the difference in the solution times is less than 10 seconds;
−: the problem was solved by both algorithms, but the LRS-based one was at least 10 slower faster;
−−: the problem was solved only by the discount algorithm.

Of 1726 interesting benchmarks, 1492 were solved by the LRS-based algorithm, and 1045 by the DISCOUNT algorithm. If these experiments were carried out as part of an interactive verification proof, a significant amount of time would have been saved by using the LRS-based algorithm.

7 Experimental Comparison of the OTTER Algorithm with and without LRS

We compared the OTTER algorithm with and without LRS using the same benchmark suites as in the previous section. The results are shown in Figure 4. Of 1318 interesting benchmarks, 1267 were solved by the LRS-based algorithm, and 589 by the standard OTTER algorithm.

To illustrate how LRS influences the proof-search statistics in terms of the share of active clauses in the kept clauses, consider the TPTP problem ANA003-1. This problem was solved by no algorithm. The following table summarises the total number of used and kept clauses.

	DISCOUNT	OTTER	LRS
used	8,191	1,967	42,050
kept	1,473,106	236,389	51,751

The DISCOUNT algorithm could process about 4 times more active clauses than the OTTER algorithm. However, it comes at a price of not performing some simplification steps. Also, the DISCOUNT algorithm kept about 6 times more clauses than the OTTER algorithm since it could not recognise that some of them are redundant w.r.t. passive clauses. The LRS-based algorithm could process about 21 times more active clauses than the OTTER algorithm and about 4 times more than the DISCOUNT algorithm. The small difference between the numbers of the kept and the active clauses shows that the calculations of reachable clauses made by LRS were quite precise.

8 Comparing LRS with Weight-Limit Based Approaches

To compare the LRS with weight-limit based approaches, we experimented with the 75 problems from the CASC-16 competition in the MIX division. The problems were run with the time limit of 5 minutes on a computer with a SPARC 233

category	++	+	=	−	−−	interesting
ALG	0	0	3	0	0	3
ANA	6	4	1	0	0	11
BOO	27	19	11	0	0	57
CAT	10	5	2	0	0	17
CID	3	0	0	0	0	3
CIV	0	3	0	0	0	3
COL	86	1	2	16	3	108
FLD	43	35	7	3	37	125
GEO	25	5	21	18	3	72
GRP	54	21	17	11	6	109
HEN	0	10	0	0	0	10
KRS	0	0	0	0	1	1
LAT	14	6	0	0	0	20
LCL	60	23	80	16	11	190
LDA	3	8	12	2	8	33
MSC	1	1	3	0	0	5
NUM	8	2	7	1	0	18
PLA	0	1	0	0	0	1
PUZ	1	1	2	0	1	5
RNG	16	5	1	0	0	22
ROB	17	0	0	0	3	20
SET	31	18	70	10	17	146
SYN	31	6	24	2	7	70
TOP	2	0	0	0	0	2
LRU	243	67	146	82	137	675
total	681	241	409	161	234	1726

Fig. 3. Comparison of the LRS-based algorithm with the DISCOUNT algorithm

category	++	+	=	−	−−	interesting
ALG	0	0	3	0	0	3
ANA	11	0	1	0	0	12
BOO	0	0	12	0	0	12
CAT	12	0	0	0	0	12
COL	20	25	32	0	3	80
FLD	88	14	2	0	7	111
GEO	62	4	9	0	0	75
GRP	25	18	26	0	6	75
LAT	6	0	4	0	0	10
LCL	152	41	25	0	5	223
LDA	12	10	2	0	0	24
MSC	0	2	2	0	0	4
NUM	17	12	3	0	1	33
PLA	1	0	0	0	0	1
PUZ	0	1	0	0	0	1
RNG	5	2	3	0	0	10
ROB	5	0	0	0	6	11
SET	67	18	40	0	5	130
SYN	3	0	63	0	0	66
TOP	2	0	0	0	0	2
LRU	241	81	59	24	18	423
total	729	228	286	24	51	1318

Fig. 4. Comparison of the LRS-based algorithm with the OTTER algorithm without LRS

MHz processor. We compare the results obtained by Vampire using the following strategies based on the OTTER algorithm: LRS, four different fixed weight limits (50, 40, 30, 20) and Incremental Weight Limit (winc). Only those 51 problems for which a proof was found by at least one strategy are considered. In the table below we give the total number of problems solved by each strategy.

strategy	LRS	50	40	30	20	winc
solved	48	45	36	27	21	33

As it can be seen from the results, the Limited Resource Strategy solves more problems than the Incremental Weight Limit Strategy or any strategy using fixed weight limit, even when the values of the weight limit are optimal for this benchmark suite. There is a problem that could only be solved by LRS (ANA002-4), and for 3 more problems the time obtained by this strategy was considerably better than by any other strategy. LRS also gives better average time than the strategy with weight limit 50, when they are compared on problems solved by both of them.

References

1. J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402, Kaiserslautern, 1995.
2. H. de Nivelle. *Bliksem 1.10 User's Manual*. MPI für Informatik, Saarbrücken, 2000.
3. P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.
4. Th. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister: High-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.
5. E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Logic Programming and Automated Reasoning. International Conference LPAR'92.*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 96–106, St.Petersburg, Russia, July 1992.
6. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.
7. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO—the CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246, 1997.
8. I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.
9. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Automated Deduction—CADE-16. 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999.

10. A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In R. Gore, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. First International Joint Conference, IJCAR 2001*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 376–380, Siena, Italy, June 2001.
11. S. Schulz. System abstract: E 0.61. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. First International Joint Conference, IJCAR 2001*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 370–375, Siena, Italy, June 2001.
12. J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. Automated Software Engineering (ASE-97)*, pages 246–254, Lake Tahoe, November 1997. IEEE Computer Society Press.
13. G. Sutcliffe. The CADE-16 ATP system competition. *Journal of Automated Reasoning*, 2000. to appear.
14. T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
15. A. Voronkov. CASC $16\frac{1}{2}$. Preprint CSPP-4, Department of Computer Science, University of Manchester, February 2000.
16. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: Spass version 1.0.0. In H. Ganzinger, editor, *Automated Deduction—CADE-16. 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 378–382, Trento, Italy, July 1999.

A Verification Approach for Distributed Abstract State Machines

Robert Eschbach

Department of Informatics, University of Kaiserslautern
Germany
eschbach@informatik.uni-kl.de

Abstract. Within verification of distributed Abstract State Machines one often has to prove properties related to the state transition level induced by partially ordered runs. This paper shows how such a transition level in form of a transition graph may be constructed. In this way the verification can be carried out directly on the transition graph. The construction itself is given as a sequential non-deterministic ASM.

1 Introduction

Distributed ASMs represent a general mathematical model of concurrent computation. In particular its notion of partially ordered runs allows as much concurrency as logically possible. Distributed ASMs have been used successfully to specify and verify distributed algorithms including the Bakery Algorithm of Lamport [1,5] and the termination detection algorithm of Dijkstra, Feijen and van Gasteren [3]. Distributed ASMs have also been used to define formal semantics for several programming (specification) languages like C [4], and more recently SDL [2].

The notion of partially ordered run is a general and adequate description of distributed computations. However, in the ASM literature often the use of partially ordered runs is avoided. We believe one reason for this can be found in the more complex structure of partially ordered runs. For example, in a partially ordered run a move is executed in several states. In general there exists no unique global state in which a move is executed. This makes verification somewhat difficult.

In order to overcome these problems and make the handling of partially ordered runs more feasible the notion of *maximal transition graph* is introduced. A maximal transition graph can be seen as a general description of all possible behaviors and can be constructed in an intuitive way. In a certain sense, the maximal transition graph contains all partially ordered runs. It can be used within the verification process to compare different runs or to reason about a single run. Some central concepts like indisputable terms, pre- and post-states of a move within a partially ordered run (cf. [5]) can be analyzed in the maximal transition graph. We believe, that the use of these kind of graphs eases the process of verification. The concept of maximal transition graphs is illustrated by some examples.

The rest of the paper is organized as follows: In section 2 we give a short introduction to distributed ASMs. In section 3 we describe maximal transition graphs and show some of their advantages.

2 Abstract State Machines

We presume the reader to be familiar with [6]. We just give a brief introduction into the concept of distributed ASMs. An ASM possesses a collection of *states*. A state can be transformed into another state by *actions* of agents. The relative ordering of actions is given by partially ordered *runs*.

2.1 States

States are basically first-order structures which contain functions and relations defined over a base set. States of an ASM have the same base set. The vocabulary associated with an ASM $\mathcal{A}$ is denoted by $\mathit{Voc}(\mathcal{A})$. Beside function or relation names a vocabulary contains declaration which are associated with names. The collection of states of $\mathcal{A}$ is denoted by $\mathit{State}(\mathcal{A})$. In the following we will not distinguish between names and their interpretations in states. External functions are completely under control of the external agents which represent the environment. In this paper we make use of internal functions only. For more information on ASM refer to [7,8].

2.2 Actions

Actions are specified by programs which are associated with agents. We distinguish between internal agents, which can change internal functions (or locations) and posses a program, and external agents, which can change external functions (or locations) only. A mathematical treatment of actions seen as special equivalence classes on update sets can be found in [7].

2.3 Runs

A partially ordered run ρ of $\mathcal{A}$ can be defined as a tuple $(M, <, A, \sigma)$ satisfying the following conditions:

1. $(M, <)$ is a partial ordering such that each segment w.r.t.$<$ induced by an element of M is finite.
2. $A : M \to \mathit{Agent}(\mathcal{A})$ is a function such that for each agent a the set $\{x : A(x) = a\}$ is linearly ordered w.r.t. $<$.
3. $\sigma : \mathit{FinSeg}(M, <) \to \mathit{State}(\mathcal{A})$, where $\mathit{FinSeg}(M, <)$ denotes the set of all finite initial segments of $(M, <)$. $\sigma(\emptyset)$ is an initial state of $\mathcal{A}$.
4. The *coherence condition*: If y is a maximal element in a finite initial segment Y of $(M, <)$ and $X = Y \setminus \{y\}$, then $A(y)$ is an agent in $\sigma(X)$, y is a move of $A(y)$ and $\sigma(Y)$ is obtained from $\sigma(X)$ by performing y at $\sigma(X)$.

The collection of all partially ordered runs of $\mathcal{A}$ is denoted by $\mathit{ParOrdRun}(\mathcal{A})$. Let $\rho = (M, <, A, \sigma) \in \mathit{ParOrdRun}(\mathcal{A})$ be a partially ordered run. The set of all finite initial segments of ρ is denoted by $\mathit{FinSeg}(\rho)$. Following [5] let $\mathit{Post}(t)$ be the set of all finite initial segments in which a move $t \in M$ is maximal called *post-segments*. For each $I \in \mathit{Post}(t)$, the state $\sigma(I)$ is the state obtained when $A(t)$ performs its action in the state $\sigma(I \setminus \{t\})$. Let $\mathit{Pre}(t) := \{I \mid I = J \setminus \{t\}, J \in \mathit{Post}(t)\}$ be the set of all *pre-segments*. Let $\mathit{PostState}(t)$ denote the set of all states that are associated with the post-segments of t, called *post-states* of t. Analogously, let $\mathit{PreState}(t)$ denote all *pre-states* of t.

3 Maximal Transition Graph

In this section we introduce the notion of *maximal transition graph*. In the following let $\mathcal{A}$ be a distributed deterministic ASM without environment (external agents). Furthermore we assume only infinite runs and a fixed set of agents.

3.1 Motivation

The maximal transition graph associated with $\mathcal{A}$ represent all possible behaviors on states. It can be used to analyze partially ordered runs. Due to its intuitive representation as a graph, it can be used within the verification of properties for partially ordered runs. Each partially run can be found within the maximal transition graph. In this way a comparison between partially ordered runs becomes more feasible.

Starting from the initial states all possible next states are related by an edge labeled with a name indicating the corresponding executing agent. In order to avoid 'cycles' between states we use a kind of ranking which excludes edges from higher to lower ranked states. We start with a simple example.

Example 1. We consider a distributed ASM $\mathcal{A}$ with a static set of internal agents a, b. Agent a increments x, agent b increments y, where x, y denotes constants of type Nat where Nat is interpreted within states as the set of natural numbers. Initially $x = y = 0$ holds. There are no external agents.

```
a: x := x + 1
b: y := y + 1
```

In figure 1 (for simplicity ranking is not shown) one can see parts of the maximal transition graph associated with ASM $\mathcal{A}$. Consider the execution path (b, b, a) which transforms the initial state $(0, 0) = (x, y)$ into state $(1, 2)$ and correspond to the partially ordered moves (1) depicted in figure 1. The first move of b is greater than the second move of b (presented by an arrow relating the first and the second move of b) which is in turn greater than the first move of a.
Now consider the execution paths $\{(b, b, a), (b, a, b)\}$ which correspond to the partially ordered set of moves (2) depicted in figure 1. Intuitively, the second move of b and the first move of a can be executed in any order (they are independent) after b has performed its first move. This can be seen in (2) where the

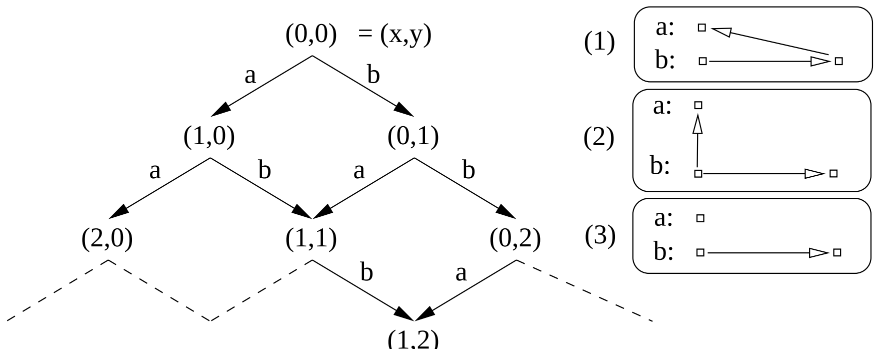


Fig. 1. Maximal Transition Graph

second move of b and the first move of a are incomparable. Note that (1) is a linearization of (2).
In figure 1 one can see that the first move of b and the first move of a may be independent, too. Consider the execution paths $\{(a, b, b), (b, b, a), (b, a, b)\}$ and the corresponding partially ordered set of moves (3) which additionally expresses this independence.

3.2 Construction

We present the construction of the maximal transition graph of ASM $\mathcal{A}$ as a sequential non-deterministic ASM $\mathcal{C}$. We forego a precise definition of the underlying vocabulary. In principal, the vocabulary with all its information can be extracted from the program. We assume the (single) executing agent to be initially in mode '*initial*'. The main program consists of three macros that will be explained in the following.

Following the definition of partially ordered runs moves of a single agent are linearly ordered. If, for example, $\mathcal{A}$ possesses an agent a then it is natural to name the i-th action of a by a_i. An *action-name* is an element from the set ActionName $= Agent(A) \times$ Nat. The *transition graph* TG will be modeled and constructed as a tuple $TG = (tg, \rightarrow)$ where $tg \subseteq$ POAState where POAState $\subseteq State(\mathcal{A}) \times$ POA, POA is a partial ordering on action names, and $\rightarrow \subseteq$ POAState $\times$ ActionName $\times$ POAState.

```
MAIN ≡
  INITIAL
  FORK
  JOIN
```

In mode '*initial*' ASM $\mathcal{C}$ chooses an initial state, initializes the transition graph $TG = (tg, <)$ and switches to mode '*fork*'.

```
Initial ≡
  if $mode = initial$ then
    $TG := (\{(s, \emptyset)\}, \emptyset)$
  $mode := fork$
```

In mode '*fork*' ASM $\mathcal{C}$ expands all maximal POA-states of TG in one step. Given a transition graph TG and an agent A, function *nextActionName* returns for a transition graph TG and agent a the next action name of a w.r.t. TG. If, for example, TG contains for a only the action names $a_0, a_1, \dots, a_i$ function *nextActionName* would return a_{i+1}. Note that in general the bounded exploration postulate (c.f. [8]) is violated. This means ASM $\mathcal{C}$ does not describe a sequential algorithm but a 'definition' for the notion of transition graph in form of a sequential construction process. Note also that in this paper we allow simultaneous extensions of sets (see the TG-assignment). Let X be a partial ordering and $y \notin X$. Then $X; \{y\}$ denotes the partial ordering in which each element of X precedes y.

```
Fork ≡
  if $mode = fork$ then
    forall $x \in max(TG)$ do
      $TG := TG \cup \{x \xrightarrow{\alpha(a)} (state(a), seg(a)) : a \in Agent(\mathcal{A})\}$
  $mode := join$
  where
    $\alpha(a) = nextActionName(TG, a)$
    $state(a) = NextState_{\mathcal{A}}(state(x), a)$
    $seg(a) = seg(x)\ ;\ \{a_k\}$
```

Let x, y be POA-states of TG. We call x and y *joinable* in TG if x, y are maximal in TG, the states are isomorphic, and the elements of the segments coincide. The segments can be joined by means of intersection. Let z be the result of joining x and y, i.e. $seg(z) = seg(x) \cap seg(y)$. We call transition graph TG *complete* if TG does not contain joinable elements.

In mode '*join*' ASM $\mathcal{C}$ chooses two joinable elements x, y and join these elements. Joining is realized as the deletion of y and the change of $seg(x)$. This will be done as long as there are joinable elements. If there are no joinable elements, i.e. TG is complete, the mode is set to '*fork*'.

```
Join ≡
  if $mode = join$ then
    choose $x, y \in max(TG) : x \neq y \wedge state(x) \cong state(y) \wedge$
        $elements(seg(x)) = elements(seg(y))$
      $TG := TG \setminus \{y\}$
      $seg(x) := seg(x) \cap seg(y)$
    if $complete(TG)$ then $mode := fork$
```

Let $\rho = (S_k : k \in \mathrm{Nat})$ be a run of $\mathcal{C}$ and $k \in \mathrm{Nat}$ such that $\mathcal{C}$ is in S_k in mode '*fork*'. By induction over Nat one can prove that the transition graph of

S_k denoted by TG_k has the following property: The segment of each element $x \in TG_k$ induces a finite partially ordered run of $\mathcal{A}$. This run is maximal independent among all finite partially ordered runs starting from the same initial state that possesses the same final state, and for all agents the same number of action occurrences.

Example 2. Consider example 1 and the transition graph (which comes with wrong labels and without segments) depicted in Fig. 1. After joining the POA-states containing state $(1, 1)$ the joined state possesses the segment $(\{a_0, b_0\}, \emptyset)$. After joining the POA-states containing state $(1, 2)$ the joined state possesses the segment $(\{a_0, b_0, b_1\}, \{b_0 \to b_1\})$, i.e. a_0 is independent from both b_0 and b_1 (cf. segment (3) of Fig. 1) .

3.3 Applications

Maximal Independent Runs. A segment of a POA-state within the maximal transition graph induces a finite, maximal independent, partially ordered run. This property can be used to analyze and construct maximal independent runs.

Pre-/Post-states. The maximal transition graph can also be used to determine the set of pre- and post-states associated with a move t. In the example 1 above the set of pre- and post-states of the second move of b w.r.t. the partial ordered set of moves (3) depicted in figure 1 can be easily found within the maximal transition graph: pre-states of this move are related to the transitions $(0, 1)$ to $(0, 2)$ and $(1, 1)$ to $(1, 2)$. The pre-states are $\{(0, 1), (1, 1)\}$ whereas the post-states are $\{(0, 2),\ (1, 2)\}$.

Indisputable Locations. Whenever a location has the same content in all pre-states of a move we say that this location is *indisputable* for this move (cf. [5]). In the example 1 the location x (more precisely $(x, ()))$ is indisputable for all moves of a in in all partially ordered set of moves (1), (2), and (3). On the other side, the location y is indisputable for all moves of b in in all partially ordered set of moves. This can be seen directly within the programs associated with a and b. For example, the location x is completely under control of a and its content in a state completely determined by the a-predecessors within each partially ordered set of moves (1), (2), and (3).

Example 3 (Example 1 continued). We change the example 1 slightly in the following way:

```
a: {Enable: }        x := x + 1
b: {Enable: even(x)} y := y + 1
```

The program of agent b has changed. It now makes use of the predicate *even* which characterizes the even natural numbers. Moves of agent b are enabled only if the content of location x in all pre-states denotes an even number. It is quite obvious how to extend the construction process for transition graphs in order to handle such enabling conditions.

In this example the location y is still completely under control of b. But now the contents is also dependent of a-moves. This changes the indisputable portions of states which can be directly seen in the corresponding maximal transition graph.

4 Conclusions

In this paper we have shown how to construct maximal transition graphs and how to use maximal transition graphs within verification. The construction itself is given as a sequential non-deterministic ASM and consists basically of a fork operation and a join operation. The applications of maximal transition graphs to the analysis of maximal independent runs, pre- and post-states, and indisputable locations shows the usability of maximal transition graphs within verification.

Acknowledgment. I thank Yuri Gurevich and Klaus Madlener for their constructive criticism on a preliminary version this paper.

References

1. Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.
2. Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, number 1912 in LNCS. Springer, 2000.
3. Robert Eschbach. A termination detection algorithm: Specification and verification. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proc. of FM'99 - World Congress on Formal Methods in the Development of Computing Systems*, number 1709 in LNCS, pages 1720–1737, 1999.
4. Yuri Gurevich and James K. Huggins. The semantics of the c programming language. In *Selected papers from CSL'92 (Computer Science Logic)*, LNCS, pages 274–308. Springer, 1993.
5. Yuri Gurevich and Dean Rosenzweig. Partially ordered runs: A case study. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines: Theory and Applications, Proc. of International Workshop, ASM2000, Monte Verità, Switzerland*, number 1912 in LNCS, pages 131–150. Springer, March 2000.
6. Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford Univ. Press, 1995.
7. Yuri Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, 1997.
8. Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

Transformational Construction of Correct Pointer Algorithms

Thorsten Ehm

Institut für Informatik
Universität Augsburg
Ehm@Informatik.Uni-Augsburg.DE

Abstract. This paper shows how to use the transformation of Paterson and Hewitt to improve the memory and operations used in a pointer algorithm. That transformation scheme normally is only of theoretical interest because of the inefficient performance of the transformed function. However we present a method how it can be used to decrease the amount of selective updates in memory while preserving the original runtime performance. This leads to a general transformation framework for the derivation of a class of pointer algorithms.

Keywords: Paterson/Hewitt, program transformation, pointer algorithms, destructive updates

1 Introduction

Algorithms on pointer structures are often used in lower levels of implementation. Although in modern programming languages (e.g. in Java) they are hidden from the programmer, they play a significant rôle at the implementation level due to their performance. But this advantage is bought at high expense. Pointer algorithms are very error-prone and so there is a strong demand for a formal treatment and development process for pointer algorithms. There are some approaches to achieve this goal.

Several methods [2,10,11] use the wp-calculus to show the correctness of pointer algorithms. There only properties of the algorithms are proved but the algorithms are not derived from a specification. So the developer has to provide an implementation. In these approaches proving trivialities may last several pages. Butler [7] investigates how to generate imperative procedures from applicative functions on abstract trees. To achieve this he enriches the trees by paths to eliminate recursion. A recent paper by Bornat [5] shows that it is possible, but difficult to reason in Hoare logic about programs that modify data structures defined by pointers. Reynolds [17] also uses Hoare logic and tries to improve a method described in a former paper of Burstall [6] to show the correctness of imperative programs that alter linked data structures.

In [13] Möller proposed a framework based on relation algebra to derive pointer algorithms from a functional specification. He shows that the rules presented also are capable of handling more difficult multi-linked data structures like doubly-linked lists or trees. However the derived algorithms are still recursive. Our goal is to improve this method by showing how to derive imperative algorithms and so achieve a more complete calculus for transformational derivation of pointer algorithms. Based on the method by Möller a recent paper by Richard Bird [4] shows how one can derive the Schorr-Waite marking algorithm in a totally functional way.

This paper shows how to use the transformation of Paterson and Hewitt (P & H) to derive imperative pointer algorithms. To achieve this we take the recursive pointer algorithms derived from functional descriptions using the method of Möller. These are transformed via the P & H transformation scheme into an imperative version. Despite the inefficient general runtime performance of the scheme that results from P & H, we get well performing algorithms. As a side effect the amount of selective updates in memory is improved by eliminating ineffective updates that are only used to pass through the pointer structure. This is not a trivial task, because in general it is not decidable if an update really changes links of the pointer structure. Some systems do such optimization during runtime but not in such an early state of software development.

We will show how these aims can be achieved for a class of pointer algorithms that first pass through a pointer structure to find the position where they have to do some proper changes. A similar transformation scheme for a class of algorithms that not only alter but also delete links is in preparation. Be aware that we are not interested in algorithms that do not alter the link structure but only the contents of the nodes (like for example `map`). The advantage of the presented method over the previously mentioned approaches using wp-calculus or Hoare logic are apparent. All these methods provide correct algorithms. However the presented one treats a whole class of functions whereas the other methods have to be applied on every new algorithm. You also do not have to provide an implementation for a specification which is a time-consuming task. Not least, the transformational approach is more likely to be the easier one to automate.

The paper is structured as follows: Section 2 defines pointer structures and describes some rules needed for the derivation of pointer algorithms from a functional version. Section 3 presents as a running example the function *cat* that appends a list to another and explains the problem. Section 4 gives a short overview of the methods to get a tail-recursive version from a linear-recursive algorithm. The transformation scheme of Paterson and Hewitt is introduced in Section 5. Section 6 describes the evolution of a transformation scheme to derive a non recursive imperative pointer algorithm from a recursive one like the one presented in Section 3. Some applications of the scheme to lists and trees are shown in Section 7. Finally Section 8 concludes the presented methodology. Appendix A includes the theoretical basics and the proofs.

2 Pointer Structures and Operations

To make this abstract self-contained as far as possible we present a short introduction to pointer structures and how they are used in [13].

In our model a pointer structure $\mathcal{P} = (s, P)$ consists of a store P and a list of entries s. The entries of a pointer structure are addresses from a set $\mathcal{A}$ that form starting points of the modeled data structures. We assume a distinguished element $\diamond \in \mathcal{A}$ representing a terminal node (e.g. null in C or nil in Pascal). A store is a family of relations (more precisely partial maps) either between addresses or from addresses to node values in a set $\mathcal{N}_j$ such as *Integer* or *Boolean*. Each relation represents a selector on the records like e.g. *head* and *tail* for lists with functionality $\mathcal{A} \to \mathcal{N}_j$ and $\mathcal{A} \to \mathcal{A}$, respectively.

Each abstract object implemented is represented by a pointer structure (n, P) with a single entry $n \in \mathcal{A}$ which represents the entry point of the data structure such as for example the root node in a tree. For convenience we introduce the access functions

$$ptr(n, L) = n \quad \text{and} \quad sto(n, L) = L$$

We want to give only the necessary definitions of operations used in this paper. More of them and proofs can be found in [13]. The following operations on relations all are canonically lifted to families of relations. Algorithms on pointer structures stand out for altering links between elements. Such modification has to be modeled in the calculus as well. We use an update operator $|$ (pronounced "onto") that overwrites relation S by relation R:

Definition 1. $R \mid S \overset{\text{def}}{=} R \cup \overline{dom(R)} \bowtie S$

Here we have used the *domain restriction* operator $\bowtie$ which is defined as $L \bowtie S = S \cap (L \times N)$ to select a particular part of $S \subseteq \mathcal{P}(M \times N)$. The update operator takes all links defined in R and adds the ones from S that no link starts from in R. To be able to change exactly one pointer in one explicit selector we define a sort of a "mini-store" that is a family of partial maps defined by:

Definition 2. $(x \overset{k}{\to} y) \overset{\text{def}}{=} \begin{Bmatrix} \{(x, y)\} & \textit{for selector } k \\ \emptyset & \textit{otherwise} \end{Bmatrix}$

It is clear that overwriting a pointer structure with links already defined in it does not change the structure:

Lemma 1. $S \subseteq T \;\Rightarrow\; S \mid T = T$ *(Annihilation)*

To have a more intuitive notation leaned on traditional programming languages, we introduce the following selective update notation:

Definition 3. *For selector k of type $\mathcal{A} \to \mathcal{A}$*
$(n, P).k := (m, Q) \overset{\text{def}}{=} (n, (n \overset{k}{\to} m) \mid Q)$

which overwrites Q with a single link from n to m at selector k. Selection is done the same way:

Definition 4.
For selector k of type $\mathcal{A} \to \mathcal{A}$: $(n, P).k \overset{\text{def}}{=} (P_k(n), P)$
For selector k of type $\mathcal{A} \to \mathcal{N}_j$: $(n, P).k \overset{\text{def}}{=} P_k(n)$

To have the possibility to insert new (unused) addresses into the data structure we define the newrec operator. Let k range over all selectors used in the modeled data structure. Then the operator $\mathsf{newrec}(L, k : x_k)$ alters the store L

to have a new record previously not in L and each selector k pointing to x_k. So for example $\mathsf{newrec}(L, head : 3, tail : \diamond)$ returns a pointer structure (m, K) with m a new address previously not used in L and store K consisting of L united with two new links $(m \overset{head}{\to} 3)$ and $(m \overset{tail}{\to} \diamond)$. If it is clear from the context which selectors are used we only enumerate the respective components. So the previous expression becomes $\mathsf{newrec}(L, \langle 3, \diamond\rangle)$.

3 A Running Example and the Problem

In this section we want to use a functional description of list concatenation. This function serves as our running example during the derivation of the transformation scheme. We will use Haskell [3] notation to denote functional algorithms:

```
cat []     ys  =  ys
cat (x:xs) ys  =  x : cat xs ys
```

We assume that the two lists are acyclic and do not share any parts. So the following pointer algorithm can be derived by transformation using the method of [13]:

$$\begin{aligned} cat_p(m, n, L) = \mathsf{if}\ m \neq \diamond\ & \mathsf{then}\ (m, L).tail := cat_p(L_{tail}(m), n, L) \\ & \mathsf{else}\ (n, L) \end{aligned}$$

The two pointer structures (m, L) and (n, L) are representations of the two lists. Addresses m and n model the starting points, whereas L is the memory going with them. In other words m and n form links to the beginnings of two lists in memory L.

Note that this is only one candidate of possible implementations for the functionally described specification of `cat`. Because we are interested in algorithms performing minimal **destructive** updates we did not derive a persistent variant such as the standard, partially copying interpretation in functional languages. But that would also be possible.

We now have a linear recursive function working on pointer structures. But what we want is an imperative program that does not use recursion. By investigating the execution order of cat_p we can see, that cat_p calculates a term of the following form:

$$(m, L).tail := ((L_{tail}(m), L).tail := ...(n, L))$$

If you remember the definition of the := operator, this means that updates are performed from right to left.

$$(m \overset{tail}{\to} L_{tail}(m)) \mid (\ldots \mid ((L^k_{tail}(m) \overset{tail}{\to} n) \mid L) \ldots)$$

This shows that the derived algorithm uses the update operator not only to properly alter links but also to just pass through the structure while returning from the recursion. Figure 1 shows how these updates are done in the example of *cat* ($\Rightarrow$ denotes the actually altered links).

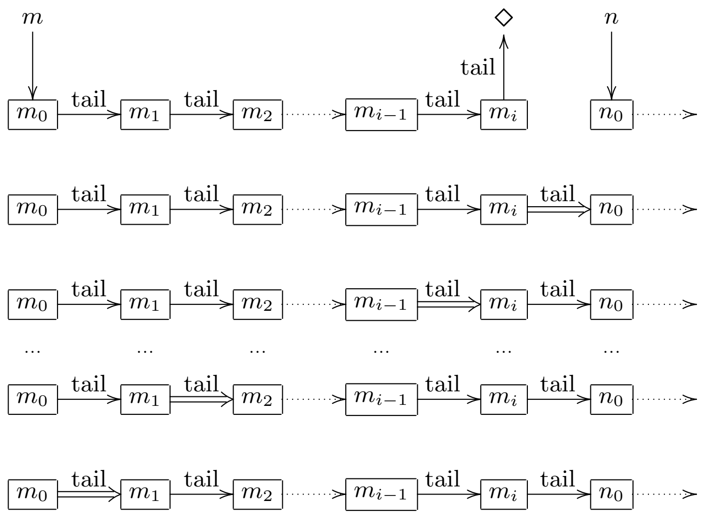


Fig. 1. Order of updates performed by *cat*

As we can see, there are several such updates that do not alter the pointer structure. For example $(m \overset{tail}{\rightarrow} L_{tail}(m))$ is already contained in L and does not change the pointer structure $(\ldots \mid ((L^k_{tail}(m) \overset{tail}{\rightarrow} n) \mid L)\ldots)$ if the previous updates do not affect this part of L. This is the case for several algorithms on pointer linked data structures, because most of them first have to scan the structure to find the position where they have to do the proper changes.

We now define the following abbreviations to get a standardized form for later transformations:

$$\begin{array}{llll} K(m,n,L) \overset{\text{def}}{=} (L_{tail}(m),n,L) & B(m,n,L) \overset{\text{def}}{=} m \neq \diamond & \phi_k(u,v) \overset{\text{def}}{=} v.k := u \\ H(m,n,L) \overset{\text{def}}{=} (n,L) & E(m,n,L) \overset{\text{def}}{=} (m,L) & \end{array}$$

Abbreviating (m,n,L) to x the derived pointer algorithm can then be written as

$$\begin{array}{ll} cat_p(x) = \mathsf{if}\ B(x) & \mathsf{then}\ \phi_{tail}(cat_p(K(x)), E(x)) \\ & \mathsf{else}\ H(x) \end{array}$$

4 From Linear via Tail Recursion to While Programs

In transformational program design the transformation of a linearly recursive function to an imperative version always has two steps: First transform the linear recursion into tail recursion. Then apply a transformation scheme [15] like the following to get a while program:

```
f(x) = if B(x)  then f(K(x))
                else H(x)
────────────────────────────⇕────────────────────────────
f(x) = var vx := x;
       while B(vx) do vx := K(vx);
       H(vx);
```

But cat_p does not have tail recursive form. So we first have to find a way to transform cat_p into the right form. There are several schemes to derive a tail recursive variant from a linear recursive function [1]. One of the most popular is to change the evaluation order of parentheses in the calculated expression. To be able to do this one needs a function ψ that fulfills the equation $\phi(\phi(r, s), t) = \phi(r, \psi(s, t))$. To find such a ψ is possible only in very rare cases of ϕ. One of these is that ϕ is associative. In this case you can choose $\psi = \phi$. Another - similar - case is to change the order of operands. Here it is necessary that $\phi(\phi(r, s), t) = \phi(\phi(r, t), s)$ or more generally you need a ψ with $\phi(\psi(r, s), t) = \psi(\phi(r, t), s)$. The previously described rules assume that ϕ is good-natured enough to satisfy one of the properties mentioned. However, our function $\phi_{tail}(u, v)$ in cat_p does not show any of these properties. Another transformation uses function inversion to calculate the parameter values from the results. Here one only has to find an inverse $\overline{K}$ of K. But the function $K(m, n, L) \overset{\text{def}}{=} (L_{tail}(m), n, L)$ in general is not invertible. So is there no way to get a tail recursive version of cat_p ?

5 The Transformation Scheme of Paterson/Hewitt

In 1970 Paterson and Hewitt presented a transformation scheme that makes it possible to transform any linear recursive function into a tail recursive one [16]. This rule normally is only of theoretical interest because of the bad runtime performance of the resulting function. P & H applied the idea of the method mentioned in Section 4 using the inverse function $\overline{K}$ to make the step from K^{i+1} to K^i, but exhaustively recalculated K^i from the start. The evolving scheme is:

```
F(x) = if B(x)  then φ(F(K(x)), E(x))
                else H(x)
──────────────────────────⇕────────────────────────── [ P & H
F(x) = G(n0, H(m0)) where
         (m0, n0) = num(x, 0)
         num(y, i) = if B(y)  then num(K(y), i + 1)
                              else (y, i)
         it(y, i) = if i ≠ 0  then it(K(y), i − 1)
                              else y
         G(i, z) = if i ≠ 0  then G(i − 1, φ(z, E(it(x, i − 1))))
                             else z
```

The function num calculates the number of iterations that have to be done until the termination condition is fulfilled, as well as the final value. These values are used by function G to change the evaluation order of the calculated term. For this, G uses the function it to iterate K to achieve the inverse $\overline{K}$ of K by doing one iteration less than has to be done for K. So G can start with the calculations done in the deepest recursion step first and then ascend from there using the inverse of K.

As we have seen, function it is only used to calculate the powers of K and we have $it(y,i) = K^i(y)$, so we can abbreviate $\phi(z, E(it(x, i-1)))$ to $\phi(z, E(K^{i-1}(x)))$ and simplify the scheme:

```
F(x) = if B(x)  then φ_k(F(K(x)), E(x))
                else H(x)
──────────────────────────↕────────────────────────── [ Paterson/Hewitt II
F(x) = G(n0, H(m0)) where
        (m0, n0) = num(x, 0)
        num(y, i) = if B(y)  then num(K(y), i + 1)
                             else (y, i)
        G(i, z) = if i ≠ 0  then G(i − 1, φ_k(z, E(K^{i−1}(x))))
                            else z
```

This certainly is only a cosmetic change, because K^{i-1} has to be calculated exactly the same way as in the original transformation scheme. But this gives the basis to future simplification, because $K^{i-1}(x)$ is only used as a parameter for E and will be eliminated in further steps.

6 Deriving a General Transformation Scheme

We now present an application of the P & H transformation scheme to pointer algorithms using the function ϕ_k to pass through a pointer data structure. The whole derivation from a more theoretical point of view including all calculations and proofs can be found in the appendix.

By investigation of function $\phi_k((m,L),(n,L)) = (n, (n \overset{k}{\rightarrow} m) \mid L)$ we can see that ϕ_k updates the link starting from m via selector k and simultaneously sets n as the new starting entry of the resulting pointer structure. It is apparent that such a restricted function can not provide the simplification we aim to achieve, namely elimination of effect-less updates. So we use the technique of generalization and introduce a more flexible function $\psi_k(l, m, (n, L)) = (l, (m \overset{k}{\rightarrow} n) \mid L)$ that handles the altered address and the resulting entry independently. With this function we are in the position to eliminate the quasi-updates that do not alter the structure but are only used for passing through the pointer structure. One can say that ψ_k “eats up” the effect-less updates of ϕ_k:

Lemma 2. *If* $(t \overset{k}{\rightarrow} v) \subseteq (v \overset{k}{\rightarrow} u) \mid U$ *then for all* s

$$\psi_k(s, t, \phi_k((u, U), (v, U))) = \psi_k(s, v, (u, U))$$

Now we return to the P & H transformation scheme. There the function G applies ϕ_k so that this lemma can be used to simplify G. We apply the lemma to all instances of G that only pass through the pointer structure. This means as long as the condition B is fulfilled we apply Lemma 2 and eliminate one application of ϕ_k. So the precondition of Lemma 2 has to hold for all those cases.

Lemma 3. *We abbreviate* $p^{(i)} \overset{\text{def}}{=} ptr(E(K^i(x)))$*. Then under the condition*

$$\forall i \in \{0, \ldots, n0\} : ptr(z) = p^{(i)} = p^{(i-1)} \vee (p^{(i)} \neq p^{(i-1)} \wedge (p^{(i-1)}, p^{(i)}) \in sto(z))$$

we can simplify $G(i,z)$ *to:*

$$G(i,z) = \mathsf{if}\ i \neq 0\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{i-1}(x))), z)\ \ \mathsf{else}\ z$$

Remembering that K is the function performing the run through the pointer structure we can express the condition in human-understandable form. The pair $(p^{(i-1)}, p^{(i)})$ consists of the values under function E of two such successive elements that are met during the pass-through via K. Now, either these are equal which means the links form a cycle and the simplification is trivial. Or they are not equal and the memory already contains the link. Then an update using these values will not change anything and can be eliminated.

With $n0 = \min\{j : \neg B(K^j(x))\}$ this is a condition that in some cases can not be checked easily. But normally one proves a more general assertion. For function *cat* for example we have acylic lists and we can show that the condition holds for all successive pairs of elements in the list.

Now that G is not recursive anymore we can instantiate the application of G with its actual parameters. The test $i \neq 0$ is only calculated once. By inspection of *num* that calculates $n0$ (the actual argument for parameter i) we see that the inequality test can be done without $n0$:

Lemma 4. $n0 \neq 0 \Leftrightarrow B(x)$

So the scheme of Paterson and Hewitt simplifies to

$$\begin{aligned} F(x) = {} & \mathsf{if}\ B(x)\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{n0-1}(x))), H(m0)) \\ & \qquad\qquad \mathsf{else}\ H(m0) \\ & \mathsf{where}\ \ (m0, n0) = num(x, 0) \\ & \qquad\qquad num(y,i) = \mathsf{if}\ B(y)\ \ \mathsf{then}\ num(K(y), i+1) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (y, i) \end{aligned}$$

A straightforward induction shows that $m0 = K^{n0}(x)$. So the calculation of $m0$ can be done simultaneously with the calculation of K^{n0-1}. This is achieved by a slightly changed pair of functions num' and num'' that replace num. For this we extend the domain of K by a special element $\triangle_x$ with $K(\triangle_x) = x$ that models an imaginary predecessor of x under K:

$$\begin{aligned} num'(y) &= \mathsf{if}\ B(y)\ \ \mathsf{then}\ num''(y) \\ &\qquad\qquad\quad \mathsf{else}\ \triangle_y \\ num''(y) &= \mathsf{if}\ B(K(y))\ \ \mathsf{then}\ num''(K(y)) \\ &\qquad\qquad\qquad \mathsf{else}\ y \end{aligned}$$

and obtain

Lemma 5. *$num'(x) = K^{n0-1}(x)$ and thus also $m0 = K(num'(x))$*

This is the basis for the following transformation:

```
F(x) = if B(x)  then ψ_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))
                else H(K(k0))
       where  k0 = num'(x)
              num'(y) = if B(y)  then num''(y)
                                 else △_y
              num''(y) = if B(K(y))  then num''(K(y))
                                     else y
──────────────────────────⇕────────────────────────── [ unfold num' and k0
F(x) = if B(x)  then ψ_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))
                else H(K(if B(x) then num''(x) else △_x))
       where  k0 = if B(x)  then num''(x)
                            else △_x
              num''(y) = if B(K(y))  then num''(K(y))
                                     else y
```

Although there is a term $E(k0)$ in the scheme the case that $E(\triangle_x)$ has to be evaluated can never be reached. So there is no need to define $E(\triangle_x)$. Now num'' is the only recursive function; it is even tail-recursive so that we are in the position to use the transformation scheme presented in Section 4 to achieve an imperative while program:

```
F(x) = if B(x)  then ψ_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))
                else H(x)
       where k0 = if B(x)  then  var vx := x
                                 while B(K(vx)) do vx := K(vx)
                                 vx
                           else △_x
──────────────────────────⇕────────────────────────── [ Simplification
F(x) = var vx := x
       if B(x)  then  while B(K(vx)) do vx := K(vx)
                      ψ_k(ptr(E(x)), ptr(E(vx)), H(K(vx)))
                else H(x)
```

The scheme that has evolved from our calculations now is:

```
F(x) = if B(x)  then φ(F(K(x)), E(x))
                else H(x)
──────────────────────────⇕────────────────────────── [ Conditions of Lemma 3
F(x) = var vx := x
       if B(x)  then  while B(K(vx)) do vx := K(vx)
                      ψ_k(ptr(E(x)), ptr(E(vx)), H(K(vx)))
                else H(x)
```

To return to our example in the previous sections we now can transform the recursive version of cat_p to an iterative program by using the derived scheme. First we check the applicability condition of our scheme abbreviating $T_i = L^i_{tail}(m)$:

$$\forall i \in \{0, \dots, \min\{j : T_j = \diamond\}\} : n = T_i = T_{i-1} \vee (T_i \neq T_{i-1} \wedge (T_{i-1}, T_i) \in L)$$

The first disjunct is not fulfilled by the assumption that the two lists do not share any parts. But the second disjunct is true by acyclicity of p. So some simplification leads us to the imperative algorithm one has in mind:

```
$cat_p(m,n,L) =$ var $(vm, vn, vL) := (m, n, L)$
                 if $m \neq \diamond$ then while $L_{tail}(vm) \neq \diamond$ do
                                               $(vm, vn, vL) := (vL_{tail}(vm), vn, vL)$
                                         $\psi_{tail}(m, vm, (vn, vL))$
                                    else $(n, L)$
──────────────────────────────$\updownarrow$────────────────────────── [ $vn, vL$ constant
$cat_p(m,n,L) =$ var $vm := m$
                 if $m \neq \diamond$ then while $L_{tail}(vm) \neq \diamond$ do $vm := L_{tail}(vm)$
                                         $(m, (vm \overset{tail}{\to} n) \mid L)$
                                    else $(n, L)$
```

This formally derived program can directly be translated into a C program by mapping the pointer algebra operations into their corresponding C equivalents. Note that the memory L remains implicit:

```
list cat(list m,list n) {
  list vm = m;
  if (m) { while (vm->tail) vm=vm->tail;
           vm->tail=n;
           return m;
  }
  else return n;
}
```

7 Further Applications

In this section we want to show that the developed scheme is applicable to several algorithms passing through a pointer-linked data structure.

7.1 Insertion into a Sorted List

In [8] several algorithms on lists are derived with the calculus presented in [13]. We choose insertion into a sorted list as a first example. The function `insert` is defined like this:

```
insert x []     = [x]
insert x (y:ys) = if x ≤ y then x:(y:ys)
                           else y:(insert x ys)
```

We can bring the derived pointer algorithm $insert_p$ into the form needed by our scheme.

```
$insert_p(m, n, L) =$
    if $n \neq \diamond \wedge L_{val}(m) > L_{head}(n)$ then $q.tail := insert_p(m, L_{tail}(n), L)$
                                                      else newrec$(L, \langle L_{val}(m), (n, L)\rangle)$
```

Now we can apply the scheme and after a simplification step achieve an imperative algorithm for insertion into a sorted list:

```
$insert_p(m, n, L) =$
    var $vn := n$
    if $(n \neq \diamond \wedge L_{val}(m) > L_{head}(n))$ then
        while $(L_{tail}(vn) \neq \diamond \wedge L_{val}(m) > L_{head}(vn))$ do $vn := L_{tail}(vn)$
        $\psi_{tail}(n, vn,$ newrec$(L, \langle L_{val}(m), (vn, L)\rangle))$
    else newrec$(L, \langle L_{val}(m), (n, L)\rangle)$
```

7.2 Insertion into a Tree

To show an example using a data structure different from lists we show how insertion into a tree can be derived from our scheme. It is nearly as easy as the other examples. We use the algorithm derived in [13] from the following functional specification:

```
ins x Empty       = Tree(Empty,x,Empty)
ins x Tree(l,y,r) = if x ≤ y then Tree(ins x l,y,r)
                             else Tree(l,y,ins x r)
```

The selectors used to model trees are val for node values as well as l and r for the left and right descendant. We abbreviate $u.val = L_{val}(u)$ and $p = (m, L)$ as before and get:

```
$ins_p\ x\ p =$ if $m = \diamond$ then newrec$(L, \langle \diamond, x, \diamond \rangle)$
                  else if $x \leq m.val$ then $p.l := ins_p\ x\ p.l$
                                         else $p.r := ins_p\ x\ p.r$
```

The algorithm can be transformed into the form needed by our scheme with the help of the conditional operator _ ? _ : _ as used in several programming languages.

```
$ins_p\ x\ p =$ if $m \neq \diamond$ then $\phi_{(x \leq m.val?l:r)}(ins_p\ x\ (x \leq m.val?p.l : p.r)), p)$
                  else newrec$(L, \langle \diamond, x, \diamond \rangle))$
```

Now we can use our scheme to achieve an imperative algorithm for insertion into a tree.

```
$ins_p\ x\ (m, L) =$
    var $vm := m$
    if $m \neq \diamond$ then while $(h := x \leq vm.val?L_l(vm) : L_r(vm)) \neq \diamond$ do $vm := h$
                   $\psi_{(x \leq vm.val?l:r)}(m, vm, \mathsf{newrec}(L, \langle\diamond, x, \diamond\rangle))$
    else newrec$(L, \langle\diamond, x, \diamond\rangle)$
```

Here we have used an assignment inside the condition of the while loop. Otherwise the algorithm would have to use the conditional operator `_ ? _ : _` twice or introduce two new while loops. But we do not think this would make the algorithm more readable.

8 Conclusion

We have shown how the transformation of Paterson and Hewitt can be used to construct imperative algorithms on pointer-linked data structures. Although the transformation of Paterson and Hewitt normally is only of theoretical interest because of its very bad runtime behaviour, well-performing algorithms are derived. This leads to a general methodology for the derivation of pointer algorithms.

By the example algorithm for insertion into a tree it can be seen, that there is a need for more sophisticated schemes based on the presented one. It also is likely that algorithms changing more than one link such as deletion from a list can be treated the same way. For this, one has to divide the job into several parts altering only one link, applying the scheme and afterwards putting the parts together.

Further research will investigate this and other starting points to complete the methodology. Also a (semi-)automatic system checking the side-conditions and so supporting the developer of such algorithms is under development.

References

1. F.L. Bauer, H. Wössner: *Algorithmic Language and Program Development*, Springer, Berlin, 1984
2. A. Bijlsma: *Calculating with pointers.* Science of Computer Programming **12**, Elsevier 1989, 191–205
3. R. Bird: *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall Press, 1998
4. R. Bird: *Unfolding Pointer Algorithms*, to appear in Journal of Functional Programming, available from:
   `http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications`
5. R. Bornat: *Proving pointer programs in Hoare logic.* Proceedings of MPC 2000, Ponte de Lima , LNCS 1837, Springer 2000, 102–126
6. R. Burstall: *Some techniques for proving correctness of programs which alter data structures.* In B. Meltzer and D. Michie eds, Machine intelligence 7, Edinburgh University Press, 1972, 23–50
7. M. Butler: *Calculational derivation of pointer algorithms from tree operations.* Science of Computer Programming **33**, Elsevier 1999, 221–260
8. T. Ehm: *Case studies for the derivation of pointer algorithms.* to appear

9. C. A. R. Hoare: *Proofs of correctness of data representations.* Acta Informatica **1**, 1972, 271–281
10. J.M. Morris: *A general axiom of assignment.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 25–34
11. J.M. Morris: *Assignment and linked data structures.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 35–51
12. B. Möller: *Towards pointer algebra.* Science of Computer Programming **21**, Elsevier, 1993, 57–90
13. B. Möller: *Calculating with pointer structures.* In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP WG2.1 Working conference, Le Bischenberg. Chapman & Hall 1997, 24–48
14. B. Möller: *Calculating with acyclic and cyclic lists.* In A. Jaoua, G. Schmidt (eds.): Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet. Information Sciences — An International Journal **119**, 1999, 135–154
15. H. Partsch: *Specification and transformation of programs. A formal approach to software development.* Monographs in Computer Science. Springer, 1990
16. M.S. Paterson, C.E. Hewitt: *Comparative Schematology.* Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Massachuset, USA, 1970
17. J.C. Reynolds: *Intuitionistic reasoning about shared mutable data structures.* In: Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, Palgrave, 2000

A Proofs

Here we show the proofs and calculations skipped in Section 6.
Proof of Lemma 2:

$$\begin{aligned}
&\psi_k(s,t,\phi_k((u,U),(v,U)))\\
=\ &\psi_k(s,t,(v,(v \overset{k}{\to} u) \mid U))\\
=\ &(s,(t \overset{k}{\to} v) \mid ((v \overset{k}{\to} u) \mid U))\\
\overset{\text{ann}}{=}\ &(s,(v \overset{k}{\to} u) \mid U)\\
=\ &\psi_k(s,v,(u,U))
\end{aligned}$$

The equality labeled with *ann* holds by annihilation if $(t \overset{k}{\to} v) \subseteq (v \overset{k}{\to} u) \mid U$. So the following conditions arise (We can restrict the calculation to k, the only selector used here):

$$\begin{aligned}
&(t \overset{k}{\to} v) \subseteq (v \overset{k}{\to} u) \mid U\\
\Leftrightarrow\ &\{(t,v)\} \subseteq \{(v,u)\} \cup \overline{dom(\{(v,u)\})} \bowtie U\\
\Leftrightarrow\ &(t,v) \in \{(v,u)\} \vee (t,v) \in \overline{\{v\}} \bowtie U\\
\Leftrightarrow\ &(t=v \wedge v=u) \vee (t \neq v \wedge (t,v) \in U)\\
\Leftrightarrow\ &(t=u=v) \vee (t \neq v \wedge (t,v) \in U) \qquad (*)
\end{aligned}$$

Proof of Lemma 3: We use Lemma 2 and induction over i:

$i = 0$**:** $G(0, z) = z$
$i \to i + 1$**:**

$$
\begin{array}{ll}
 & G(i+1, z) \\
= & \quad \{\!\![\text{ definition of } G \text{ and } i+1 \neq 0 \ ]\!\!\} \\
 & G(i, \phi_k(z, E(K^i(x)))) \\
= & \quad \{\!\![\text{ induction hypothesis }]\!\!\} \\
 & \mathsf{if}\ i \neq 0\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{i-1}(x))), \phi_k(z, E(K^i(x)))) \\
 & \qquad\qquad \mathsf{else}\ z \\
= & \quad \{\!\![\text{ (*) and lemma 2 }]\!\!\} \\
 & \mathsf{if}\ i \neq 0\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^i(x))), z) \\
 & \qquad\qquad \mathsf{else}\ z
\end{array}
$$

Proof of Lemma 4:

$$
\begin{array}{ll}
 & n0 \neq 0 \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } n0 \ ]\!\!\} \\
 & snd(num(x, 0)) \neq 0 \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } num \ ]\!\!\} \\
 & snd(\mathsf{if}\ B(x)\ \mathsf{then}\ num(x, 1)\ \mathsf{else}\ (x, 0)) \neq 0 \\
\Leftrightarrow & \quad \{\!\![\text{ propagation of } snd \text{ and } \neq 0 \ ]\!\!\} \\
 & (B(x) \wedge snd(num(x, 1)) \neq 0) \vee (\neg B(x) \wedge \mathtt{false}) \\
\Leftrightarrow & \quad \{\!\![\ snd(num(x, i)) = 0 \Leftrightarrow i = 0 \ ]\!\!\} \\
 & B(x)
\end{array}
$$

Proof of Lemma 5: First let $h(x, y, i) = K^{snd(num(y,i))-1}(x)$. So we can derive a recursive definition of h by:

$$
\begin{array}{ll}
 & h(x, y, i) \\
= & \quad \{\!\![\text{ definition of } h \ ]\!\!\} \\
 & K^{snd(num(y,i))-1}(x) \\
= & \quad \{\!\![\text{ definition of } num \ ]\!\!\} \\
 & K^{snd(\mathsf{if}\ B(y)\ \mathsf{then}\ num(K(y),i+1)\ \mathsf{else}\ (y,i))-1}(x) \\
= & \quad \{\!\![\text{ propagation of } K,\ snd \text{ and } -1 \ ]\!\!\} \\
 & \mathsf{if}\ B(y)\ \ \mathsf{then}\ K^{snd(num(K(y),i+1))-1}(x) \\
 & \qquad\qquad \mathsf{else}\ K^{snd(y,i)-1}(x)
\end{array}
$$

$=$ $\{\!\![$ definition of h $]\!\!\}$

$\textsf{if } B(y) \ \textsf{then } h(x, K(y), i+1)$
$\qquad\quad \textsf{else } K^{i-1}(x)$

By computational induction we now show

$$num''(K^i(x)) = h(x, K^{i+1}(x), i+1) \qquad (*)$$

$num''(K^i(x))$

$=$ $\{\!\![$ definition of num'' $]\!\!\}$

$\textsf{if } B(K(K^i(x))) \ \textsf{then } num''(K(K^i(x)))$
$\qquad\qquad\qquad \textsf{else } K^i(x)$

$=$ $\{\!\![$ induction hypothesis $]\!\!\}$

$\textsf{if } B(K^{i+1}(x)) \ \textsf{then } h(x, K(K^{i+1}(x)), (i+1)+1)$
$\qquad\qquad\quad \textsf{else } K^{(i+1)-1}(x)$

$=$ $\{\!\![$ recursive definition of h $]\!\!\}$

$h(x, K^{i+1}(x), i+1)$

Using the special case $h(x, K(x), 1) = num''(x)$ of this equivalence, we can show the claim:

$K^{n0-1}(x)$

$=$ $\{\!\![$ definition of $n0$ $]\!\!\}$

$K^{snd(num(x,0))-1}(x)$

$=$ $\{\!\![$ definition of h $]\!\!\}$

$h(x, x, 0)$

$=$ $\{\!\![$ recursive definition of h $]\!\!\}$

$\textsf{if } B(x) \ \textsf{then } h(x, K(x), 1)$
$\qquad\quad \textsf{else } \Delta_x$

$=$ $\{\!\![$ by (*) $]\!\!\}$

$\textsf{if } B(x) \ \textsf{then } num''(x)$
$\qquad\quad \textsf{else } \Delta_x$

$=$ $\{\!\![$ definition of num' $]\!\!\}$

$num'(x)$

A Theoretical Foundation of Program Synthesis by Equivalent Transformation

Kiyoshi Akama[1], Hidekatsu Koike[2], and Hiroshi Mabuchi[3]

[1] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
akama@cims.hokudai.ac.jp,
[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
koke@cims.hokudai.ac.jp,
[3] Iwate Prefectural University, 152-52 Sugo, Takizawa, Iwate, 020-0173 Japan,
mabu@soft.iwate-pu.ac.jp

Abstract. Equivalent transformation (ET) is useful for synthesis and transformation of programs. However, it is not so clear what semantics should be preserved in synthesis and transformation of programs in logic and functional programming, which come from the disagreement of computation models (inference or evaluation) and equivalent transformation. To overcome the difficulty, we adopt a new computation model, called equivalent transformation model, where equivalent transformation is used not only for program synthesis, but also for computation. We develop a simple and general foundation for computation and program synthesis, and prove the correctness of ET-based program synthesis.

1 Introduction

Equivalent transformation is one of the most important methods for program synthesis and transformation [6]. For instance, in logic programming [5], a predicate definition consisting of first order formulas or definite clauses is transformed equivalently into a (more efficient) logic program (i.e., a set of definite clauses) by using unfolding, folding, goal replacement, and other transformations. In functional programming, a function definition is transformed equivalently into a (more efficient) functional program by using unfolding, folding, tupling, and other transformations.

In this paper we develop a theoretical foundation of program synthesis by equivalent transformation (ET), define basic concepts, and prove correctness of **ET-based program synthesis**. This theory should contain the following items:

1. Definition of specification, computation, and programs,
2. Definition of correctness of computation and programs (with respect to a specification),
3. Relation between computation and equivalent transformation,
4. Proof of correctness of programs obtained by equivalent transformation.

It should be noted that such a theory has not been fully established in the existing theories of logic or functional programming. For instance, computation is regarded not as equivalent transformation but as logical inference (resolution) in logic programming and it is not clear which declarative semantics should be preserved in equivalent transformation for correct program synthesis.

Instead of developing a theory in the existing frameworks of logic or functional computation model, we adopt a new computation model, called **equivalent transformation model** [1]. In the equivalent transformation model, equivalent transformation is used not only for program synthesis but also for computation. This enables us to make a simple and general foundation for computation and program synthesis.

Our theory consists mainly of the following sections:

1. Theory of Computation,
2. Separated Descriptions,
3. Program Synthesis by Equivalent Transformation.

Outline of the theory is summarized as follows.

1. Theory of Computation

General computation theory is developed on base structures called **representation systems**, where each **description** is associated with its **meaning**. A **rewriting rule** is used to transform a **description** into another **description**. A **rewriting rule** is called an **equivalent transformation rule (ET rule for short)** iff it preserves **meanings** of **descriptions**. **Computation** is defined as transformation of **descriptions**. **Computation** is **correct** iff it is equivalent transformation of **descriptions**. A **program** is a set of **rewriting rules**. A **program** is **correct** iff it consists of only **ET rules**. A **correct program** produces **correct computation**.

2. Separated Descriptions

We introduce a subclass of **representation systems**, called **separated representation systems**, where (1) a **description** consists of two parts, a **d-description** and a **q-description** and (2) **meaning** of a **description** (d, q) depends only the **q-description** q and meaning of the **d-description** d. Such description is called a **separated description**. A description in logic programming is regarded as a separated description consisting of a predicate definition (= **d-description**) and a query (= **q-description**).

We also assume that only **q-descriptions** are changed by **rewriting rules** and the **d-descriptions** are left unchanged in computation.

3. Program Synthesis by Equivalent Transformation

A **specification** is determined by a pair of a **d-description** and a set of **q-descriptions**. A **rewriting-rule-set generator** is a mapping that associates each **d-description** with a set of **rewriting rules**. An **ET-rule-set generator** is a **rewriting-rule-set generator** that gives, for each **d-description**, a set of **ET rules**.

A correct program (= a set of **ET rules**) with respect to a specification consisting of a **d-description** d is obtained in the following two phases:

1. equivalent transformation of the given **d-description** d into another **d-description** d', and
2. the new **d-description** d' is mapped by an **ET-rule-set generator** into a program (= a set of **ET rules**).

2 Theory of Computation

2.1 Representation System

A **representation system** is a triple $\langle Des, v, Meg\rangle$ of two sets, Des and Meg, and a mapping v from Des to Meg. Each element of Des and Meg are called a **description** and a **meaning**, respectively. A relation $\sim$ on Des is defined by
$des_1 \sim des_2 \iff v(des_1) = v(des_2)$.
Obviously $\sim$ is an equivalence relation.

Example 1. Let $\mathcal{P}(\Delta)$ be the powerset of the set of all definite clauses on an alphabet Δ. Let $\mathcal{G}$ be the set of all ground atoms on the alphabet Δ. Let Des_1 be $\mathcal{P}(\Delta)$, Meg_1 the powerset $2^{\mathcal{G}}$, and v_1 a mapping $\mathcal{M}$ from $\mathcal{P}(\Delta)$ to $2^{\mathcal{G}}$ that determines so called the "declarative semantics" of P for each P in $\mathcal{P}(\Delta)$. Then $\langle Des_1, v_1, Meg_1\rangle$ is a representation system.

2.2 Problem Formalization Based on Meaning

A problem on a representation system $\langle Des, v, Meg\rangle$ is a triple
$\alpha = \langle des, \pi, \mathcal{K}\rangle$,
where des is a description in Des, $\mathcal{K}$ is a set, and π is a mapping from Meg to $\mathcal{K}$. The problem
$\alpha = \langle des, \pi, \mathcal{K}\rangle$
on a representation system $\langle Des, v, Meg\rangle$ requires to find the element k in $\mathcal{K}$ determined by $k := \pi(v(des))$.

Example 2. Let C_1, C_2, and C_3 be the following definite clauses, where *app* means *append*.

C_1 $initial(X, Z) \leftarrow app(X, Y, Z).$
C_2 $app([\,], Y, Y) \leftarrow.$
C_3 $app([A|X], Y, [A|Z]) \leftarrow app(X, Y, Z).$

Let C_q be a definite clause
$yes \leftarrow initial([1, 2], [1, 2, 3, 4]).$
Let π_1 be a mapping from $2^{\mathcal{G}}$ to $\{yes, no\}$ that is defined by

$$\begin{aligned} \pi_1(G) &= yes \qquad && yes \in G, \\ &= no && yes \notin G. \end{aligned}$$

Then $\alpha_1 = \langle\{C_1, C_2, C_3, C_q\}, \pi_1, \{yes, no\}\rangle$ is a problem to judge whether $[1, 2]$ is a first part of $[1, 2, 3, 4]$. Since $\mathcal{M}(\{C_1, C_2, C_3, C_q\})$ includes the *yes* atom, the answer of Problem α_1 is "*yes*".

2.3 Transformation by Rewriting Rules

A **rewriting rule** on Des is defined as a subset of $Des \times Des$. A description des_1 is rewritten into a description des_2 by a rewriting rule r, denoted by $des_1 \overset{r}{\to} des_2$, iff $(des_1, des_2) \in r$. Let Rw be a set of rewriting rules. A description des_1 is rewritten into a description des_2 by Rw, denoted by $des_1 \overset{Rw}{\to} des_2$, iff there is a rewriting rule r in Rw such that $des_1 \overset{r}{\to} des_2$.

One way to solve Problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$ on a representation system $\langle Des, v, Meg \rangle$ is shown.

Algorithm A(Rw)
Let Rw be a set of rewriting rules on Des.
Input: a problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$.
Output: an element in $\mathcal{K}$.

1. Assume that a problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$ on $\langle Des, v, Meg \rangle$ is given.
2. Transform des in Des by repeated application of rewriting rules in Rw into des' in Des.
   $des = des_1 \overset{Rw}{\to} des_2 \overset{Rw}{\to} des_3 \overset{Rw}{\to} \cdots \overset{Rw}{\to} des_n = des'$
3. Calculate $k := \pi(v(des'))$ and return k.

Example 3. For instance,
r1: $initial(\&X, \&Z) \to app(\&X, \#Y, \&Z)$.
is used to replace an *initial* atom in the body of a clause with an *app* atom that satisfies the following conditions [1] :

- The first and the third arguments of the *app* atom are the same as the first and the second arguments of the *initial* atom, respectively.
- The second argument of the *app* atom should be a "new" variable that does not appear in other part of the clause.

This is a rewriting rule (i.e., it is a subset of $Des_1 \times Des_1$, where Des_1 is in Example 1) since it determines the set of all pairs (des, des') of descriptions in Des_1 such that des is rewritten into des' by $r1$ by replacement of an *initial* atom in a clause in des with an *app* atom.

Example 4. Let P_1 be a set of rewriting rules:
$P_1 = \{r1, r2, r3\}$,
where $r1$ is in Example 3 and other two rules $r2$ and $r3$ are given as follows.

r2: $app([\,], \&Y, \&Z) \to \{\&Y = \&Z\}$.
r3: $app([\&A|\&X], \&Y, \&Z) \to \{\&Z = [\&A|\#W]\},$
$\qquad\qquad\qquad\qquad app(\&X, \&Y, \#W)$.

The process of rewriting the definite clause C_q is as follows.

[1] See appendix and [2] for explanation of rules.

$yes \leftarrow initial([1,2],[1,2,3,4]).$	
$yes \leftarrow app([1,2],Y,[1,2,3,4]).$	by Rule r1
$yes \leftarrow app([2],Y,[2,3,4]).$	by Rule r3
$yes \leftarrow app([\,],Y,[3,4]).$	by Rule r3
$yes \leftarrow .$	by Rule r2

Since the last description des' obtained by the above rewriting contains the clause $yes \leftarrow$, it follows that $\pi_1(v(des')) = yes$ and "yes" is obtained as the output of Algorithm A({r1, r2, r3}).

2.4 Problem Solving Based on Equivalent Transformation

An **equivalent transformation** (ET) rule is defined as a rewriting rule r that satisfies $v(des_1) = v(des_2)$ for all pairs (des_1, des_2) in r. In order to transform elements in Des equivalently, ET rules are used.

Theorem 1. *If the set Rw consists only of ET rules, the Algorithm A(Rw) gives a correct answer for any problem* α.

Example 5. Three rewriting rules (r1, r2, and r3) in Example 4 are ET rules. For instance, r1 is an ET rule since r1 works in the same way as unfolding at an *initial* atom. Since $r1$, $r2$, and $r3$ are ET rules, Problem α_1 is correctly solved by using Algorithm $A(\{r1, r2, r3\})$.

2.5 Program Consisting of ET Rules

A set P of rewriting rules can be regarded as a **program**, since P determines a (possibly nondeterministic) algorithm based on the Algorithm A(Rw). If a program P consists only of ET rules, then computation by P is correct, i.e., a correct answer for Problem α is obtained.

Example 6. Let P_2 be a set of ET rules:
$P_2 = \{r4, r5\}$,
where $r4$ and $r5$ are given as follows.

r4: $initial([\&A|\&X], \&Z) \rightarrow \{\&Z = [\&A|\#W]\},$
$\qquad\qquad\qquad\qquad\qquad initial(\&X, \#W).$
r5: $initial([\,], \&Z) \rightarrow \langle true \rangle.$

P_2 is a correct program for α_1.

3 Separated Descriptions

3.1 Separated Representation Systems

Let C be a definite clause. Let $head(C)$ and $body(C)$ denote, respectively, the head atom of C and the set of all atoms in the body of C. Let $\mathcal{S}$ be the set of all substitutions on Δ. Let R_a and R_b be sets of predicates. A clause C is from R_a to R_b iff all predicates of atoms in $body(C)$ are in R_a and the predicate of $head(C)$ is in R_b.

The description $\{C_1, C_2, C_3, C_q\}$ in Problem α_1 in Example 2 consists of two parts, $d_0 = \{C_1, C_2, C_3\}$ and $q_0 = \{C_q\}$. The meaning of the description $d_0 \cup q_0 = \{C_1, C_2, C_3, C_q\}$ satisfies

$\mathcal{M}(d_0 \cup q_0) = \mathcal{M}(d_0) \cup t(\mathcal{M}(d_0), q_0)$,

where

$t(G, q) = \{h \mid C \in q,\ \theta \in \mathcal{S},\ h = head(C\theta) \in \mathcal{G},\ body(C\theta) \subseteq G\}$.

Hence there is a mapping m such that

$\mathcal{M}(d_0 \cup q_0) = m(\mathcal{M}(d_0), q_0)$.

This example motivates us to introduce the following definition.

A **separated representation system** is a six-tuple

$\langle Ds, Qs, w, Ms, m, Meg\rangle$

of two sets Ds and Qs for descriptions, two sets Ms and Meg for meanings, a mapping w from Ds to Ms, and a mapping m from $Ms \times Qs$ to Meg.

Assume that $\langle Ds, Qs, w, Ms, m, Meg\rangle$ is a separated representation system. If we define a set Des as $Ds \times Qs$, and a mapping v from Des to Meg by

$v(des) = m(w(d), q)$

for all $des = (d, q)$ in Des, then $\langle Des, v, Meg\rangle$ is obviously a representation system, which is called the **associated representation system** of the separated representation system $\langle Ds, Qs, w, Ms, m, Meg\rangle$.

A separated representation system $\langle Ds, Qs, w, Ms, m, Meg\rangle$ is always identified with its associated representation system.

Hereafter we assume that we are given a separated representation system $\Gamma = \langle Ds, Qs, w, Ms, m, Meg\rangle$.

Example 7. Let R_1 and R_2 be mutually disjoint sets of predicates:

$R_1 = \{initial, app, equal, rev\}$,

$R_2 = \{yes, ans\}$.

Let Ds_7 be the powerset of the set of all definite clauses from R_1 to R_1, Qs_7 the powerset of the set of all definite clauses from R_1 to R_2, Ms_7 the powerset of the set of all ground atoms on R_1, and Meg_7 the powerset of the set of all ground atoms on $R_1 \cup R_2$. Let w_7 be a mapping from Ds_7 to Ms_7 such that

$w_7(d) = \mathcal{M}(d)$.

Let m_7 be a mapping from $Ms_7 \times Qs_7$ to Meg_7 defined by

$m_7(G, q) = G \cup t(G, q)$,

where t is defined in Section 3.1. Then

$\Gamma_7 = \langle Ds_7, Qs_7, w_7, Ms_7, m_7, Meg_7\rangle$

is a separated representation system. Let $\langle Des_7, v_7, Meg_7\rangle$ be the associated representation system of the separated representation system Γ_7. Then

$v_7((d, q)) = \mathcal{M}(d \cup q)$

for all $(d, q) \in Des_7$.

3.2 Rewriting Rules and ET Rules

A subset r of $Qs \times Qs$ and an element d in Ds determines a rewriting rule r':
$r' = \{((d, q), (d, q')) \mid (q, q') \in r\}$,
which is called the **associated rewriting rule** of r with respect to d. A subset r of $Qs \times Qs$ is called a **rewriting rule** on Qs. A rewriting rule r on Qs is called an **ET rule** with respect to an element d in Ds iff the associated rewriting rule r' of r with respect to d is an ET rule. Obviously a rewriting rule r on Qs is an **ET rule** with respect to d in Ds iff
$m(w(d), q) = m(w(d), q')$
for all $(q, q') \in r$. A description (d, q) in $Ds \times Qs$ is equivalently transformed into (d, q') in $Ds \times Qs$ by an ET rule r on Qs with respect to d in Ds.

4 Program Synthesis by Equivalent Transformation

4.1 Specification

Let $\Gamma = \langle Ds, Qs, w, Ms, m, Meg \rangle$ be a separated representation system. A **specification** on Γ is a pair (d, Q) of d in Ds and a subset Q of Qs. A specification (d, Q) on Γ requires a program to answer all problems (d, q) such that $q \in Q$.

4.2 ET-Rule-Set Generator

Let $\Gamma = \langle Ds, Qs, w, Ms, m, Meg \rangle$ be a separated representation system. A mapping g from Ds to the powerset of the set of all rewriting rules on Qs is called a **rewriting-rule-set generator** on Γ. For all d in Ds, a rewriting-rule-set generator g on Γ determines a set $g(d)$ of rewriting rules on Qs.

A rewriting-rule-set generator g on Γ is called an **ET-rule-set generator** on Γ iff, for all d in Ds, each element in $g(d)$ is an ET rule with respect to d.

Example 8. d_0 is a set $\{C_1, C_2, C_3\}$ in Section 3.1. Let g_0 be a rewriting-rule-set generator that generates, for each predicate, one rewriting rule that is similar to the set of all flatten clauses of the definition clauses for the predicate [2]. Then, $g_0(d_0) = \{r1, r6\}$, where

$$\begin{aligned} \text{r6:}\quad \mathit{app}(\&P, \&Q, \&R) &\to \{\&P = [\,], \&Q = \&R\}; \\ &\to \{\&P = [\#A|\#X], \&R = [\#A|\#Z]\}, \\ &\quad\ \mathit{app}(\#X, \#Y, \#Z). \end{aligned}$$

4.3 Obtaining ET-Rules by Equivalent Transformation

Theorem 2. *Assume that g is an ET-rule-set generator on Γ. If two elements d and d' in Ds satisfies $w(d) = w(d')$, then $g(d')$ is a set of ET rules with respect to d.*

[2] See [3] for precise definition and the correctness of rules that are obtained by g_0.

Let $\Gamma = \langle Ds, Qs, w, Ms, m, Meg\rangle$ be a separated representation system. Since w is a mapping from Ds to Ms, $\langle Ds, w, Ms\rangle$ is a representation system. A relation $\sim$ on Ds is defined by

$d_1 \sim d_2 \iff w(d_1) = w(d_2)$.

Obviously $\sim$ is an equivalence relation. According to the general definitions of rewriting rules and ET rules, a rewriting rule r on Ds is an ET rule on Ds iff $w(d) = w(d')$ for all (d, d') in r.

Algorithm B(Rw, g)

Let Rw be a set of rewriting rules on Ds, and g a rewriting-rule-set generator on Γ.

Input: a specification (d_0, Q) on Γ

Output: a set of rewriting rules on Qs.

1. Transform d_0 into d_n by repeated application of rewriting rules in Rw, i.e.,
   $d_0 \overset{Rw}{\rightarrow} d_1 \overset{Rw}{\rightarrow} \cdots \overset{Rw}{\rightarrow} d_n$.
   Rewriting rules may be applied any finite times ($n \geq 0$) as long as they are applicable.
2. From d_n, obtain a set of rewriting rules $g(d_n)$ by using the rewriting-rule-set generator g.

Theorem 3. *If all rewriting rules in Rw are ET rules and g is an ET-rule-set generator on Γ, then the set of rewriting rules obtained by Algorithm B(Rw, g) is a set of ET rules with respect to d_0.*

Example 9. Assume that the set $d_0 = \{C_1, C_2, C_3\}$ in Section 3.1 is transformed equivalently by using unfolding and folding into a new set $d_2 = \{C_{11}, C_{12}, C_2, C_3\}$, where

C_{11} $\mathit{initial}([\,], Z) \leftarrow .$

C_{12} $\mathit{initial}([A|X], [A|Z]) \leftarrow \mathit{initial}(X, Z)$.

From d_2 we obtain a set $g_0(d_2) = \{r6, r7\}$ of ET rules using the ET-rule-set generator g_0.

r7: $\mathit{initial}(\&P, \&R) \rightarrow \{\&P = [\,], \&R = \#Z\};$
$\rightarrow \{\&P = [\#A|\#X], \&R = [\#A|\#Z]\},$
$\mathit{initial}(\#X, \#Z).$

The set $\{r6, r7\}$ is a correct program with respect to d_0.

5 Concluding Remarks

A theoretical basis for program synthesis by equivalent transformation has been developed. Given a specification (d_0, Q) on a separated representation system $\Gamma = \langle Ds, Qs, w, Ms, m, Meg\rangle$, a program is obtained by transforming d_0 equivalently into d_n and by mapping d_n using an ET-rule-set generator g. An element d in Ds is transformed in **program synthesis** preserving meaning $w(d)$ of d, while an element q in Q is transformed in **computation** preserving meaning $m(w(d), q)$ of (d, q). This theory can be applied to many declarative programs including logic and functional programs.

References

1. Akama, K., Shigeta, Y., and Miyamoto, E., A Framework of Problem Solving by Equivalent Transformation of Logic Program, J. Japan Soc. Artif. Intell., Vol.12, No.2, pp.90–99 (1997).
2. Akama, K., Nantajeewarawat, E., and Koike, H., A Class of Rewriting Rules and Reverse Transformation for Rule-Based Equivalent Transformation, Proc. 2nd International Workshop on Rule-Based Programming, Firenze, Italy, 2001.
3. Nantajeewarawat, E., Akama, K., and Koike, H., Expanding Transformation as a Basis for Correctness of Rewriting Rules, Tech. Report, Hokkaido University, 2001.
4. Futamura, Y., Partial Evaluation of Computation Process - an Approach to a Compiler-Compiler. Systems. Computers. Controls. Vol.25. pp.45–50 (1971).
5. Lloyd, J. W., Foundations of Logic Programming, Second edition, Springer-Verlag, 1987.
6. Pettorossi, A. and Proietti, M., Transformation of Logic Programs: Foundations and Techniques, *The Journal of Logic Programming*, Vol.19/20, pp.261–320, (1994).

A Appendix

A **rewriting rule** for transforming a set of definite clauses consists of meta-atoms, where a meta-atom is an atom that includes &-variables and #-variables instead of usual variables. An &-variable is a variable that begins with & (such as $\&X$) and can be replaced with an arbitrary (usual) term. A #-variable is a variable that begins with # (such as $\#C$) and can be replaced with an arbitrary (usual) variable.

A rewriting rule is of the form ($n \geq 0$):

$$\begin{aligned}
&\langle rulename \rangle\text{:} \\
&H, \{Cs\} \rightarrow \{Es_1\}, Bs_1; \\
&\qquad\quad\ \rightarrow \{Es_2\}, Bs_2; \\
&\qquad\qquad\quad \cdots \\
&\qquad\quad\ \rightarrow \{Es_n\}, Bs_n.
\end{aligned}$$

Where $\langle rulename \rangle$ is the name of a rule, H is a meta-atom, Cs is a (possibly empty) sequence of executable meta-atoms, Es_i are (possibly empty) sequences of executable meta-atoms, and Bs_i are (possibly empty) sequences of meta-atoms. H is called the *head*, Cs the *applicability condition part*, each Es_i an *execution part*, and each pair of Es_i and Bs_i a *body* of the rule. The $\langle rulename \rangle$, the condition part, and each execution part are optional.

Assume that we are given an atom b in the body of a definite clause C. A rewriting rule of this form is applicable to the atom b iff the head H matches the atom b by a substitution θ (i.e., $H\theta = b$) and $Cs\theta$ is true (by the given evaluator). When the rule is applied to a clause C at an atom b, C produces less than or equal to n clauses. Each new clause is obtained, after $Es_i\theta$ is executed successfully (by the given evaluator) with an answer substitution σ, by replacing $b\sigma$ in $C\sigma$ with $Bs_i\theta\sigma$.

Equivalent Transformation by Safe Extension of Data Structures

Kiyoshi Akama[1], Hidekatsu Koike[2], and Hiroshi Mabuchi[3]

[1] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
akama@cims.hokudai.ac.jp,
[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
koke@cims.hokudai.ac.jp,
[3] Iwate Prefectural University, 152-52 Sugo, Takizawa, Iwate, 020-0173 Japan,
mabu@soft.iwate-pu.ac.jp

Abstract. Equivalent transformation has been proposed as a methodology for providing programs with appropriate data structures. For instance, logic programs which use lists are transformed into equivalent programs that use difference-lists. However lists and difference-lists are both usual terms and in this sense no new data structures are introduced in the transformation. Since logic programming has fixed data structure called terms, no one can develop theoretical foundations for introducing new data structures into programs as far as only logic programs are discussed. In this paper we develop a theoretical foundation of equivalent transformation that introduces new data structures. We introduce a parameter Γ for data structures, by which many languages with different data structures are characterized. By changing this parameter (say from Γ_1 to Γ_2) we can discuss data structure change for programs. We define a concept of safe extension of data structures, and prove that the meaning of a program on a data structure is preserved by safe extension of the data structure.

1 Introduction

Equivalent transformation is one of the most important methods for improving efficiency of programs [9]. Declarative programs such as logic and functional programs may be improved into more efficient ones by using unfolding, folding, goal replacement, tupling, and other transformations.

It is well known that efficiency of computation depends on data structures. In case of procedural programming languages, it is taken for granted to adopt better data structures for efficient computation [10]. Data structures are also important in logic and functional computation model.

Equivalent transformation has been proposed as a methodology for providing programs with appropriate data structures. For instance, several methods for the transformation of logic programs which use lists into equivalent and more efficient programs that use difference-lists have been proposed in the literature [5,8,11].

However, it should be noted that lists and difference-lists are included in usual terms and in that sense no new data structures are introduced in the transformation. Theoretical foundation of such transformation is exactly the same as usual equivalent transformation of programs by unfolding, folding, goal replacement, and other transformation techniques.

Extension of data structures in this paper is totally different. While the above methods transform logic programs into other logic programs with data structure unchanged in the sense that they both use terms, in our theory new data structures that are not included in the original domain are introduced in the transformation. Typical examples for such new data structures are class variables and domain variables, which are not included in usual term domain. Expressive power and efficiency is improved by using class variables based on sort hierarchy [1,2]. Many constraint satisfaction problems cannot be solved within practical time in Prolog, while constraint logic programs with domain variables solve them far more efficiently than Prolog [6].

In this paper we develop a theoretical foundation of equivalent transformation that introduces new data structures, based on which we can often make programs more efficient. For instance, in the case of class-variables, improvement of efficiency of programs is obtained by the following equivalent transformations:

1. (ET_1) introduction of class variables,
2. (ET_2) transformation of programs using class variables.

In the case of domain-variables, programs are similarly improved by the following two steps:

1. (ET_1) introduction of domain variables,
2. (ET_2) transformation of programs using domain variables.

In both cases, introduction of new data structures (ET_1) is essential to further transformation (ET_2).

The main purpose of the paper is to formalize the first equivalent transformation (ET_1). Since logic programming has a fixed data structure called terms, no one can develop theoretical foundations for introducing new data structures into programs as far as only logic programs are discussed. It is crucial to introduce a parameter Γ for many data structures, by which many languages with different data structures are characterized. Mathematical structure called a **specialization system** is introduced as a parameter for specifying data structures in problem description. Declarative descriptions are also introduced on specialization systems. A declarative description d on a specialization system Γ is associated with its meaning $\mathcal{M}(\Gamma, d)$. Equivalent transformation is a transformation of (Γ, d) preserving $\mathcal{M}(\Gamma, d)$. Then, we have two typical equivalent transformations for a pair (Γ, d) of a declarative description d and a specialization system Γ:

1. ET_1: $(\Gamma_1, d) \to (\Gamma_2, d)$ Change from Γ_1 into Γ_2 with d unchanged,
2. ET_2: $(\Gamma, d_1) \to (\Gamma, d_2)$ Change from d_1 into d_2 with Γ unchanged.

We define **extension** and **safe extension** of specialization systems to formulate introduction of new data structures. We also prove the correctness of equivalent transformation by safe extension of specialization systems, i.e.,
$\mathcal{M}(\Gamma_1, d) = \mathcal{M}(\Gamma_2, d)$
for any Γ_1 and Γ_2 such that Γ_2 is a safe extension of Γ_1.

2 Efficiency Improvement by Class Variables

2.1 A Vehicle Problem

In order to discuss improvement of efficiency by introducing new data structures, consider a sample problem :

> Bicycles and cars are vehicles. There are two bicycles; "bike1" and "bike2." There are three cars; "car1", "car2", and "mycar." Vehicles have tires. Cars have doors. I own "mycar." Find all objects that have tires and doors and that I own.

2.2 Formalization by a Logic Program

The above problem is formalized easily by a logic program P_L and a query q_L as follows:

$$\begin{aligned} P_L = \{ \; & \mathit{vehicle}(X) \leftarrow \mathit{bicycle}(X). \\ & \mathit{vehicle}(X) \leftarrow \mathit{car}(X). \\ & \mathit{bicycle}(\mathit{bike}1) \leftarrow. \\ & \mathit{bicycle}(\mathit{bike}2) \leftarrow. \\ & \mathit{car}(\mathit{car}1) \leftarrow. \\ & \mathit{car}(\mathit{car}2) \leftarrow. \\ & \mathit{car}(\mathit{mycar}) \leftarrow. \\ & \mathit{has}(Z, \mathit{tires}) \leftarrow \mathit{vehicle}(Z). \\ & \mathit{has}(Z, \mathit{doors}) \leftarrow \mathit{car}(Z). \\ & \mathit{own}(i, \mathit{mycar}) \leftarrow. \\ & \mathit{answer}(Y) \leftarrow \mathit{has}(Y, \mathit{tires}),\, \mathit{has}(Y, \mathit{doors}),\, \mathit{own}(i, Y). \; \}. \\ q_L = \; & \mathit{answer}(Z). \end{aligned}$$

2.3 Formalization by a Sorted Logic Program

Since the sample problem is related to concepts such as vehicles, cars, and bicycles, it is natural to formalize it on the basis of a concept hierarchy. One such formalization follows:

$$\begin{aligned} P_S = \{ \; & \mathit{vehicle} \supset \; \mathit{bicycle} \;\mid\; \mathit{car} \\ & \mathit{bicycle} \ni \; \mathit{bike}1 \;\mid\; \mathit{bike}2 \\ & \mathit{car} \ni \; \mathit{car}1 \;\mid\; \mathit{car}2 \;\mid\; \mathit{mycar} \end{aligned}$$

$has(A : vehicle, tires) \leftarrow .$
$has(B : car, doors) \leftarrow .$
$own(i, mycar) \leftarrow .$
$answer(Y) \leftarrow has(Y, tires), has(Y, doors), own(i, Y). \}.$
$q_S = answer(Z).$

In the program P_S, "vehicle", "bicycle", and "car" are classes, and "bike1", "bike2", "car1", "car2", and "mycar" are instances. Capital letters such as Y and Z are variables. The expressions "A : *vehicle*" and "B : *car*" are sorted terms (class variables), representing a vehicle A and a car B, respectively.

The sorted program P_S consists of three declarations and four clauses. The knowledge of concept hierarchy is represented more compactly by declarations in P_S than by clauses in P_L. The four clauses in P_S are also more concise than the corresponding clauses in P_L due to the use of class variables.

2.4 Towards a Common Theoretical Foundation

We already know that the second program P_S is more efficient than the first one P_L to answer the same queries (q_L and q_S) [1,2]. When given the first program P_L, we want to transform P_L into P_S for efficient computation. Two questions now arise:

How is P_L transformed into P_S?

How is the transformation justified?

In this paper, such questions will be solved in a general setting.

In the previous subsections, two formalizations, (P_L, q_L) and (P_S, q_S), have been discussed separately without common theoretical foundation. In the subsequent sections, we will introduce a general framework, where the two formalizations in the previous subsections will be re-formalized in a single framework. This is essential to establish efficiency improvement by equivalent transformation.

3 Specialization Systems

3.1 Definition of Specialization Systems

Definition 1. *A specialization system is a four-tuple $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ of three sets $\mathcal{A}$, $\mathcal{G}$ and $\mathcal{S}$, and a mapping $\mu : \mathcal{S} \rightarrow partial\_map(\mathcal{A})$ that satisfies the following requirements, where $partial\_map(X)$ is the set of all partial mappings on X.*

1. $\forall s_1, s_2 \in \mathcal{S}, \exists s \in \mathcal{S} : \mu(s) = \mu(s_2) \circ \mu(s_1)$.
2. $\exists s \in \mathcal{S}, \forall a \in \mathcal{A} : \mu(s)(a) = a$.
3. $\mathcal{G} \subseteq \mathcal{A}$.

Elements of $\mathcal{A}$ are called atoms. Elements of $\mathcal{G}$ are called ground atoms. Elements of $\mathcal{S}$ are called specializations.

When there is no danger of confusion, elements in $\mathcal{S}$ are regarded as partial mappings over $\mathcal{A}$ and the following notational convention is used. Each element in $\mathcal{S}$ is identified as a partial mapping on $\mathcal{A}$, and the application of such a partial mapping is represented by postfix notation. For example, $\theta \in \mathcal{S}$ and $\mu(\theta)(a)$ are denoted respectively by $\theta \in \mathcal{S}$ and $a\theta$.

3.2 Examples of Specialization Systems

Example B. A s-term is either a constant such as *bike*2 and a pure variable[1] such as X. A s-atom is an atom of the form $p(t_1, t_2, \cdots, t_n)$, where p is a predicate and each t_i is a s-term. Let $\mathcal{A}_b$ be the set of all s-atoms. Let $\mathcal{G}_b$ be the set of all ground (variable free) s-atoms. A binding θ transforms an atom in $\mathcal{A}_b$ by replacing variables with s-terms. Let $\mathcal{S}_b$ be the set of all sequences of bindings. An element s in $\mathcal{S}_b$ transforms an atom in $\mathcal{A}_b$ into another atom in $\mathcal{A}_b$ by successive application of bindings in s, which defines the mapping $\mu_b : \mathcal{S}_b \to \mathit{map}(\mathcal{A}_b)$, where $\mathit{partial\_map}(X)$ is the set of all mappings on X. Then $\Gamma_b = \langle \mathcal{A}_b, \mathcal{G}_b, \mathcal{S}_b, \mu_b \rangle$ is a specialization system.

Example D. A c-term is either a constant such as *bike*1, a pure variable such as Y, or a class variable such as $B : \mathit{vehicle}$. A class variable is a pair of a tag such as B and a class such as *vehicle*. Pure variables and class variables are called simply variables. A c-atom is an atom of the form $p(t_1, t_2, \cdots, t_n)$, where p is a predicate and each t_i is a c-term. Let $\mathcal{A}_d$ be the set of all c-atoms. Let $\mathcal{G}_d$ be the set of all ground (variable free) c-atoms.

Four kinds of basic specializations are introduced; a *variable substitution* such as (X, mycar), a *tag renaming* such as (B, C), a *class restriction* such as $(C, \mathit{bicycle})$, and a *tag substitution* such as $(B, \mathit{car}2)$.

A pure variable is specialized by a variable substitution. For example, Z is specialized to $X : \mathit{car}$ by $(Z, X : \mathit{car})$. A class variable is specialized by a tag renaming. For instance, $B : \mathit{car}$ is specialized to $C : \mathit{car}$ by (B, C). A class variable is specialized by reducing its class into its subclass. $B : \mathit{vehicle}$ is specialized to $B : \mathit{car}$ by (B, car). A class variable is specialized to a constant in its class. For instance, $B : \mathit{car}$ is specialized to *mycar* by (B, mycar).

A basic specialization transforms an atom in $\mathcal{A}_d$ into another atom in $\mathcal{A}_d$. Let $\mathcal{S}_d$ be the set of all sequences of basic specializations. An element s in $\mathcal{S}_d$ transforms an atom in $\mathcal{A}_d$ by successive application of basic specializations in s, which defines a mapping $\mu_d : \mathcal{S}_d \to \mathit{partial\_map}(\mathcal{A}_d)$. Then $\Gamma_d = \langle \mathcal{A}_d, \mathcal{G}_d, \mathcal{S}_d, \mu_d \rangle$ is a specialization system.

Note that, among these four types of basic specializations, the last two basic specializations may not be applicable to some objects. For instance, basic specialization $(B, \mathit{bicycle})$ cannot be applied to $B : \mathit{car}$. This is because *bicycle* is not a subclass of *car*. Similarly, basic specialization $(B, \mathit{car}1)$ cannot be applied to $B : \mathit{bicycle}$ since *car*1 is not an element of *bicycle*.

[1] A pure variable is an ordinary variable, but we use this term to distinguish it from a class variable.

4 Declarative Description

4.1 Declarative Description

Definition 2. *Let Γ be a specialization system $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$. A definite clause on Γ is a formula of the form:*
$H \leftarrow B_1, B_2, \cdots, B_n$
where $H, B_1, B_2, \cdots, B_n$ are atoms in $\mathcal{A}$. A declarative description on Γ is a set of definite clauses on Γ.

Let C be a definite clause on X. The head of a clause C is denoted by $head(C)$, and the set of all atoms in the body of C is denoted by $body(C)$.

A specialization $s \in \mathcal{S}$ is applicable to $a \in \mathcal{A}$ iff there exists $b \in \mathcal{A}$ such that $\mu(s)(a) = b$. When $\theta \in \mathcal{S}$ is applicable to $H, B_1, B_2, \cdots, B_n$, a definite clause $C\theta = (H\theta \leftarrow B_1\theta, B_2\theta, \cdots, B_n\theta)$ is obtained from a definite clause $C = (H \leftarrow B_1, B_2, \cdots, B_n)$. A definite clause C' is an instance of C iff there is a specialization θ such that $C' = C\theta$. A definite clause C is ground iff it consists of only ground atoms. A ground instance of a definite clause C is a ground definite clause that is an instance of C.

Let P be a declarative description on Γ. The set of all ground instances of definite clauses in P is denoted by $Gclause(P)$.

Example 1. P_L is a declarative description on Γ_b. P_S is a declarative description on Γ_d.

4.2 Meaning of a Declarative Description

For a declarative description P, the meaning $\mathcal{M}(P)$ is defined by
$\mathcal{M}(P) = \bigcup_{n=0}^{\infty} [T_P]^n(\emptyset)$,
where T_P is a mapping on the powerset $2^{\mathcal{G}}$, which mapps a subset of $\mathcal{G}$ into another subset of $\mathcal{G}$:
$T_P(x) = \{head(C) \mid body(C) \subseteq x, C \in Gclause(P)\}$.
Since $Gclause(P)$ and T_P depend on the specialization system Γ, $\mathcal{M}(P)$ also depends on Γ. Hereafter when Γ should be specified explicitly, $\mathcal{M}(P)$ will be denoted by $\mathcal{M}(\Gamma, P)$.

5 Increase of Meaning by Extension

We consider two specialization systems Γ_1 and Γ_2, and discuss the relationship between the two meanings of a declarative description on the two specialization systems.

5.1 Extension of Specialization Systems

Consider a mapping $\mu : \mathcal{S} \to partial\_map(\mathcal{A}_d)$. The mapping μ determines a subset μ' of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$ uniquely by

$\mu' = \{(s, a, b) \mid s \in \mathcal{S}, a \in \mathcal{A}, b \in \mathcal{A},\ \mu(s)(a) = b\}$.

It is obvious that the mapping that determines μ' from μ is one-to-one. In this paper μ will be identified with μ', thus μ will be regarded as a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$, which is convenient for discussion of the relation between two specialization systems.

Definition 3. *Let Γ_1 and Γ_2 be specialization systems:*
$\Gamma_1 = \langle \mathcal{A}_1, \mathcal{G}_1, \mathcal{S}_1, \mu_1 \rangle$,
$\Gamma_2 = \langle \mathcal{A}_2, \mathcal{G}_2, \mathcal{S}_2, \mu_2 \rangle$.
Γ_2 is an **extension** *of Γ_1 (or Γ_1 is a* **partial specialization system** *of Γ_2), iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$, $\mathcal{G}_1 \subseteq \mathcal{G}_2$, $\mathcal{S}_1 \subseteq \mathcal{S}_2$, and $\mu_1 \subseteq \mu_2$.*

Note that μ_1 and μ_2 are regarded, respectively, as subsets of $\mathcal{S}_1 \times \mathcal{A}_1 \times \mathcal{A}_1$ and $\mathcal{S}_2 \times \mathcal{A}_2 \times \mathcal{A}_2$, and that $\mu_1 \subseteq \mu_2$ iff any elements (θ, a, b) in μ_1 are included also in μ_2.

Example 2. Γ_d is an extension of Γ_b.

5.2 Inclusion Relation of Meaning

The following theorem states that the meaning of a declarative description on a specialization system increases by extension of the specialization system.

Theorem 1. *Assume that Γ_2 is an extension of Γ_1. If P is a declarative description on Γ_1, then P is also a declarative description on Γ_2, and*
$\mathcal{M}(\Gamma_1, P) \subseteq \mathcal{M}(\Gamma_2, P)$.

6 Preservation of Meaning by Safe Extension

6.1 Safe Extension

Definition 4. *Let Γ_1 and Γ_2 be specialization systems:*
$\Gamma_1 = \langle \mathcal{A}_1, \mathcal{G}_1, \mathcal{S}_1, \mu_1 \rangle$,
$\Gamma_2 = \langle \mathcal{A}_2, \mathcal{G}_2, \mathcal{S}_2, \mu_2 \rangle$.
Γ_2 is a **safe extension** *of Γ_1 iff the following conditions are satisfied.*

1. *Γ_2 is an extension of Γ_1.*
2. *$\mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$.*
3. *For any finite subset X of $\mathcal{A}_1$ and for any specialization θ in $\mathcal{S}_2$ such that $X\theta \subseteq \mathcal{G}$, there is a specialization σ in $\mathcal{S}_1$ such that $x\theta = x\sigma$ for any $x \in X$.*

Example 3. Γ_d is a safe extension of Γ_b.

6.2 Preservation of Meaning

The following theorem states that the meaning of a declarative description on a specialization system is preserved by safe extension of the specialization system.

Theorem 2. [Safe-extension Theorem] *Assume that Γ_2 is a safe extension of Γ_1. If P is a declarative description on Γ_1, then P is also a declarative description on Γ_2 and*
$\mathcal{M}(\Gamma_1, P) = \mathcal{M}(\Gamma_2, P)$.

7 Application of the Theorem

Based on the safe-extension theorem, we discuss the equivalence between the declarative description P_L on the specialization system Γ_b and the declarative description P_S on the specialization system Γ_d.

Since Γ_d is a safe extension of Γ_b, we have, by theorem 2,
$\mathcal{M}(\Gamma_b, P_L) = \mathcal{M}(\Gamma_d, P_L)$.

The three predicates *vehicle*, *bicycle*, and *car*, which are defined in the declarative description P_L on the specialization system Γ_d, can be represented more compactly by the following declarative description P_M, when we use the structure of the specialization system Γ_d.

$$\begin{aligned} P_M = \{\ & vehicle(A : vehicle) \leftarrow. \\ & bicycle(A : bicycle) \leftarrow. \\ & car(A : car) \leftarrow. \\ & has(Z, tires) \leftarrow vehicle(Z). \\ & has(Z, doors) \leftarrow car(Z). \\ & own(i, mycar) \leftarrow. \\ & answer(Y) \leftarrow has(Y, tires),\ has(Y, doors),\ own(i, Y).\ \ \} \end{aligned}$$

Then, we have
$\mathcal{M}(\Gamma_d, P_L) = \mathcal{M}(\Gamma_d, P_M)$.
Moreover, when we remove the body of *has* clause in the declarative description P_M on the specialization system Γ_d by unfolding, and delete three clauses, (*vehicle*, *bicycle*, and *car* clauses), we obtain a declarative description P_S on the specialization system Γ_d and
$\mathcal{M}(\Gamma_d, P_M) - H = \mathcal{M}(\Gamma_d, P_S)$,
where, H is the meaning of the declarative description consisting of the three clauses; *vehicle*, *bicycle*, and *car*.

Thus we have
$\mathcal{M}(\Gamma_b, P_L) - H = \mathcal{M}(\Gamma_d, P_S)$.
Therefore,
$\{g \mid answer(g) \in \mathcal{M}(\Gamma_b, P_L)\}$
and
$\{g \mid answer(g) \in \mathcal{M}(\Gamma_d, P_S)\}$
are the same, and the representation change from 2.2 to 2.3 can be justified by the theory in this paper.

8 Conclusion

Each theory in computer science has been usually discussed on one specific language with fixed data structure. Necessity of unified theoretical foundation for many languages with different data structures has not been widely recognized. In this paper, we first defined specialization systems and declarative descriptions on specialization systems. A problem was formalized as a pair of a specialization system and a declarative description. The concept of specialization systems characterizes data structures for many languages, and is essential to develop a theory for introducing new data structures into problem descriptions.

If we change a specialization system Γ_1 into Γ_2 with a declarative description d left unchanged, (Γ_1, d) is transformed into (Γ_2, d). If Γ_2 is a safe extension of Γ_1, the transformation from (Γ_1, d) to (Γ_2, d) is an equivalent transformation, i.e., $\mathcal{M}(\Gamma_1, d) = \mathcal{M}(\Gamma_2, d)$. This theory can be applied to many examples of efficiency improvement by introducing new data structures, including class-variable examples and domain-variable examples.

References

1. Aït-Kaci, H. and Nasr, R., *LOGIN: A Logic Programming Language with Built-In Inheritance*, The Journal of Logic Programming, 3 (1986).
2. Akama, K., *PAL: An Extended Prolog with Inheritance Hierarchy*, information processing society of Japan, Vol.28 No.4 pp.27-34 (1987).
3. Akama, K., Nomura, Y., and Miyamoto, E., *Semantic Interpretation of Natural Languages by Program Transformation*, Computer Software, Vol.12, No.5, pp.45–62 (1995).
4. Akama, K., Shigeta, Y., and Miyamoto, E., *A Framework of Problem Solving by Equivalent Transformation of Logic Program*, J. Japan Soc. Artif. Intell., Vol.12, No.2, pp.90–99 (1997).
5. Brough, D. R. and Hogger, C. J., *Compiling Associativity into Logic Programs*, *The Journal of Logic Programming*, Vol.4, pp.345–359 (1987).
6. Hentenryck, V., *Constraint Satisfaction in Logic Programming*, The MIT Press (1989).
7. Lloyd, J. W., *Foundations of Logic Programming*, Second edition, Springer-Verlag (1987).
8. Marriot, K. and Sondergaad, H., *Difference-List Transformation for Prolog*, *New Generation Computing*, Vol.11, pp.125–177 (1993).
9. Pettorossi, A. and Proietti, M., *Transformation of Logic Programs: Foundations and Techniques*, *The Journal of Logic Programming*, Vol.19/20, 1994, pp.261–320.
10. Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall (1976).
11. Zhang, J. and Grant, P. W., *An Automatic Difference-List Transformation Algorithm for Prolog*, *Proc. 1988 European Conference on Artificial Intelligence (ECAI'88)*, pp.320–325 (1988).

Semantics and Transformations in Formal Synthesis at System Level*

Viktor Sabelfeld, Christian Blumenröhr, and Kai Kapp

Institute of Computer Design and Fault Tolerance, Karlsruhe University
{sabelfel,blumen,kai.kapp}@ira.uka.de   http://goethe.ira.uka.de/fsynth

Abstract. In formal synthesis methodology, circuit implementations are derived from specifications by means of elementary logical transformation steps, which are performed within a theorem prover. In this approach, additionally to the circuit implementation, the proof that the result is a correct implementation of a given specification is obtained automatically. In the paper, we formally describe the functional semantics of system specifications in higher order logic. This semantics builds the basis for formal synthesis at system level. Further, theorems for circuit optimisation at this level are proposed.

1 Introduction

The most critical question in circuit synthesis is the correctness of the design: how can one guarantee the correctness of the automatically generated circuit implementation with regard to a given specification? The synthesis programs are too big and too complex to prove their correctness using the available software verification tools.

In formal synthesis, the circuit implementation is obtained from the specification by application of elementary transformation rules which are formulated in higher order logic and proved as theorems in a theorem prover. The correctness of a transformation means a mathematical relation between the source and the result. If such correct transformations are used during the synthesis process, then, as a result, not only the circuit implementation, but also a proof of the correctness of this implementation with regard to the specification ("correctness by construction") is obtained.

In the formal synthesis, most approaches deal with the synthesis of digital systems at lower levels of abstraction [9], or with synthesis of pure data-flow descriptions at the algorithmic level [7].

Besides formal synthesis, there are approaches which can be summarized by the term "transformational design". They also claim to fulfill the paradigm of "correctness by construction": The synthesis process is also based on correctness-preserving transformations. However, the transformations are proved by paper

* This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-3.

& pencil and afterwards implemented in a complex software program. It is implicitly assumed that an automatic synthesis process is correct by definition. But there is no guarantee that the transformations have been implemented correctly and therefore, the correctness of the synthesis process cannot be regarded as proved. A successfully applied approach for synthesis at the system level that falls into this category is described in [5].

2 Circuit Descriptions at the Algorithmic Level

We have developed the language *Gropius* [1] to describe the circuit specifications and implementations in our approach to formal synthesis.

The BNF below describes the syntactic structure of DFG-terms (acyclic **D**ata **F**low **G**raphs). They represent non-recursive programs that always terminate.

$$\begin{array}{rl}
\textit{var\_structure} ::= & \textit{variable}\ [\text{“:”}\ \textit{type}\ ] \mid \text{“(”}\ \textit{var\_structure}\ \text{“,”}\ \textit{var\_structure}\ \text{“)”} \\
\textit{expression} ::= & \textit{var\_structure} \mid \textit{constant}\ [\text{“:”}\ \textit{type}\ ] \\
& \mid \text{“(”}\ \textit{expression}\ \text{“,”}\ \textit{expression}\ \text{“)”} \mid \text{“(”}\ \textit{expression}\ \textit{expression}\ \text{“)”} \\
\textit{DFG-term} ::= & \text{“}\lambda\text{”}\ \textit{var\_structure}\ \text{“.”} \\
& [\ \text{“}\mathsf{let}\text{”}\ \textit{var\_structure} = \textit{expression}\ \text{“}\mathsf{in}\text{”}\ ]\ \textit{expression}
\end{array}$$

A *condition* is a DFG-term, which produces a boolean output for an input of arbitrary type. We have fixed the following syntax in Gropius for program terms (*P-term*) which are used for circuit descriptions at algorithmic level:

$$\begin{array}{rl}
\textit{P-term} ::= & \text{“}\mathsf{DEFINED}\text{”}\ \textit{DFG-term} \mid \text{“}\mathsf{WHILE}\text{”}\ \textit{condition}\ \textit{P-term} \\
& \mid \textit{P-term}\ \text{“}\mathsf{SERIAL}\text{”}\ \textit{P-term} \mid \textit{P-term}\ \text{“}\mathsf{PARALLEL}\text{”}\ \textit{P-term} \\
& \mid \text{“}\mathsf{IF}\text{”}\ \textit{condition}\ \textit{P-term}\ \textit{P-term}
\end{array}$$

The P-terms have a type $\alpha \to \beta\,\mathsf{partial}$. To represent the data type of a P-term which may not terminate, the type $\alpha\,\mathsf{partial}$ has been introduced. This type extends an arbitrary type α by a new value $\perp$: $\mathsf{partial} = \perp \mid \mathsf{Def\ of}\ \alpha$.

The functions $\mathsf{resolve}$ and $\mathsf{resolve}^2$ have been introduced in order to define new functions on single and paired values of type $\alpha\,\mathsf{partial}$, respectively:

$$(\mathsf{resolve} \perp f\, a \overset{\text{def}}{=} a) \wedge (\mathsf{resolve}\,(\mathsf{Def}\ x)\, f\, a \overset{\text{def}}{=} f\, x)$$
$$\mathsf{resolve}^2\ (a : \alpha\ \mathsf{partial})(b : \beta\ \mathsf{partial})(B : \alpha \times \beta \to \gamma\ \mathsf{partial}) \overset{\text{def}}{=}$$
$$\mathsf{resolve}\ a\ (\lambda u.\ \mathsf{resolve}\ b\ (\lambda v.\ B(u,v))\ \perp)\ \perp$$

Below, we give the formal definitions for the semantics of the P-terms A SERIAL B and A $\mathsf{PARALLEL}$ B. More details about P-terms and the algorithmic level of Gropius can be found in [2].

$$(A : \alpha \to \beta\,\mathsf{partial})\ \mathsf{SERIAL}\ (B : \beta \to \gamma\,\mathsf{partial}) \overset{\text{def}}{=} (\lambda x.\, \mathsf{resolve}\ (A\ x)\ B\ \perp)$$
$$(A : \alpha \to \beta\,\mathsf{partial})\ \mathsf{PARALLEL}\ (B : \alpha \to \gamma\,\mathsf{partial}) \overset{\text{def}}{=} (\lambda x.\, \mathsf{resolve}^2(A\, x)(B\, x)\ \mathsf{Def})$$

The first definition says that (A SERIAL B) is a function which, for a given $x : \alpha$, delivers the value $\perp$, if $A\ x = \perp$, and it delivers the value $B\ y$, if $A\ x = \mathsf{Def}\ y$.

Similarly, (A $\mathsf{PARALLEL}$ B) is defined as a function which, for a given $x : \alpha$, delivers $\perp$, if $A\ x = \perp$ or $B\ x = \perp$, and it delivers the value $\mathsf{Def}\,(y, z)$, if $A\ x = \mathsf{Def}\ y$ and $B\ x = \mathsf{Def}\ z$.

An algorithmic description expresses the functional dependencies of outputs on the inputs of the circuit. It does not take time aspects into account. During high-level synthesis the algorithmic description is mapped to a register transfer (RT) level structure. To bridge the gap between these two different abstraction levels one has to determine how the circuit communicates with its environment. Therefore, as a second component of the circuit representation an interface description is required. The interface description defines the temporal signal behavior at the circuit interface and specifies exactly the communication of the circuit with the environment. In contrast to most existing approaches, we strictly separate between the algorithmic and the interface description. We provide a set of interface patterns, of which the designer can select one. More details about interface patterns can be found in [2,3].

3 Circuit Descriptions at the System Level

Below we give the definitions for the syntax and semantics of system descriptions called here System-structures or S-structures for short. In our approach to synthesis at system level, all the processes interact via a fixed communication scheme which is label-based and inspired by higher order Petri nets [6]. Our process corresponds to a Petri net transition with some places as its inputs and outputs. "Firing" means here removing the input labels, performing some calculation and delivering the result as a new label to the output places.

At the system level, processes communicate via channels. They have a fixed structure and consist of three signals: a signal *data* of arbitrary type α, and two boolean control signals *data_valid* and *ready*. Each channel has a fixed direction, which is defined by the direction of the signal *data*. The *data_valid*-signal goes to the same direction whereas the *ready*-signal goes to the opposite direction.

In channels, communication is performed via handshake. Let us consider a channel from a process A to a process B. A signals via *data_valid* that there is a label with some data being on *data*. Process B signals via *ready* that it is ready to read the next label. Whenever both *data_valid* and *ready* become true, the communication takes place, i.e. the label is moved from A to B.

Four kinds of processes will be used as components in S-structures: DFG-term and P-term based processes, K-processes (see Section 4) and S-calls. The DFG- and P-term based processes are essentially nothing else but circuit descriptions at the algorithmic level. The only difference is that the functional description is combined with a special interface pattern for the system level. Both DFG-term based processes and P-term based processes have a single input channel and a single output channel. K-processes are a sort of a glue logic for building arbitrary S-structures and for managing the communication (German: Kommunikation) between other processes. They may have an arbitrary (but fixed) number of input and output signals and are used to spread, combine, buffer, delay or synchronize signals. Finally, S-calls are nothing else but procedure calls. One can define (non-recursive) S-structures and give them arbitrary names. S-processes allow the use of hierarchical circuit descriptions in Gropius and reduce the complexity of the synthesis process. The syntax of S-structures is as follows:

```
     interface ::= "(" channel { "," channel } ")"
   DFG-process ::= "Dfg_proc" DFG-term interface
     P-process ::= "P_proc" program interface
        S-call ::= structure-name interface
     S-process ::= DFG-process | P-process | K-process | S-call
     K-process ::= ("Double" | "Join" | "Synchronize" | "Split" | "Fork"
                   | "Choose" | "Sink" | "Source" constant ) interface
   S-structure ::= "∃" { channel } "." S-process { "∧" S-process }
  S-definition ::= structure-name interface "=" S-structure
```

On system level, in contrast to all other Gropius specifications, S-structures are described relationally, as it is desirable and necessary to permit cycles at the system level.

4 K-Processes

We have defined eight elementary K-processes in Gropius: Double, Join, Split, Synchronize, Fork, Choose, Source, and Sink. In K-processes, no calculations on data labels are performed. Labels are solely moved according to the label based communication pattern presented in the previous section.

The K-process Double duplicates the input label, Join combines two separate labels into a single paired label, Split is the inverse process to Join.

The K-process Synchronize collects two labels. As soon as both successor processes are ready, both labels are given over simultaneously.

The K-process Fork delivers the first component d of an incoming label (d, b) of type $\alpha \times \mathsf{bool}$ to one of the output channels, depending on whether the second component of the label is false or true.

The K-process Choose has two input channels in_1 and in_2 for labels of type $\alpha \times \mathsf{bool}$ and one output channel *out* for labels of type α. It is the only process, which can fire even if not all inputs are ready; moreover, the channels in_1 and in_2 can never be ready simultaneously. Choose delivers to its successor the first value d of the label (d, b) it became either from in_1 or in_2. There are two different states, Ready_1 and Ready_2, in which Choose can fire. Initially, Choose is in the state Ready_1. In the state Ready_i, only channel in_i is ready. Every time a data label (d, b) occurs, firing delivers the label d to the output channel *out*; it leaves the state unchanged, if $b = \mathsf{T}$, or changes it to Ready_{3-i}, if $b = \mathsf{F}$.

The process Source C is a source in the data flow which does not have input channels and yields the constant C every time when the output channel is ready to receive a signal. The process Sink has one input channel but no output channels and implement a sink of a signal. It is always ready to read a label from their input channel. That means, a label can always leave the predecessor process and move to Sink where it disappears afterwards.

5 Functional Semantics of S-Structures

In [3], we have defined the behavioral semantics and the equivalence relation for S-structures, which consider the temporal input-output signal behavior. But, an exact definition of the signal flow in time, as given in the behavioral semantics, is often not desirable. To take the purely functional aspects of the system descriptions into account, we will define the functional semantics and the functional equivalence relation for S-structures. Informally, the functional semantics of an S-structure fixes only the sequences of the output signals for the given sequences of input signals. We represent these (finite or infinite) signal sequences in the theorem prover HOL as values of the type abstraction $\alpha\,\mathsf{signal}$, which are functions f of type $(\mathsf{num} \rightarrow \alpha\,\mathsf{partial})$ satisfying the following *signal condition*: $\mathsf{Is\_signal}\ f \overset{\text{def}}{=} \forall n.\,(f\,n = \bot \Rightarrow \mathsf{Idle}\,f\,n)$, where $\mathsf{Idle}\,f\,n \overset{\text{def}}{=} \forall k.\,(n \leq k \Rightarrow (f\,k = \bot))$. Here, the value $\bot$ can be interpreted as the absence of any signal value. Using the type definition package by Melham [8] we have defined a representation function $\mathsf{from\_signal} : \alpha\,\mathsf{signal} \rightarrow (\mathsf{num} \rightarrow \alpha)$ and its inverse abstraction function $\mathsf{to\_signal}$, so that the following theorem holds:

$$\begin{aligned}&\vdash (\forall x : \alpha\,\mathsf{signal}.\ \mathsf{to\_signal}\,(\mathsf{from\_signal}\,x) = x) \wedge \\ &\quad (\forall f : \mathsf{num} \rightarrow \alpha\,\mathsf{partial}.\ \mathsf{Is\_signal}\,f = (\mathsf{from\_signal}(\mathsf{to\_signal}\ f) = f))\end{aligned}$$

An example of a signal is the signal $\mathsf{blank} \overset{\text{def}}{=} \mathsf{from\_signal}\,(\lambda n.\,\bot)$. In order to build new signals from existing ones we introduce two operators Δ (for single value signals), and Δ^2 (for paired value signals):

$$\begin{aligned}
&\Delta\ (x : \alpha\ \mathsf{signal})\ (A : \alpha \rightarrow \beta\ \mathsf{partial})\ (0 : \mathsf{num}) \overset{\text{def}}{=} \mathsf{resolve}\ (\mathsf{from\_signal}\ x\ 0)\ A\ \bot \\
&\Delta\ x\ A\ (\mathsf{SUC}\ n) \overset{\text{def}}{=} \mathsf{resolve}^2\ (\Delta\ x\ A\ n)(\mathsf{from\_signal}\ x\ (\mathsf{SUC}\ n))\ (A \circ \mathsf{SND}) \\
&\Delta^2\ (x : \alpha\ \mathsf{signal})(y : \beta\ \mathsf{signal})(B : \alpha \times \beta \rightarrow \gamma\ \mathsf{partial})\ (0 : \mathsf{num}) \overset{\text{def}}{=} \\
&\qquad \mathsf{resolve}^2\ (\mathsf{from\_signal}\ x\ 0)(\mathsf{from\_signal}\ y\ 0)\ B \\
&\Delta^2\ x\ y\ B\ (\mathsf{SUC}\ n) \overset{\text{def}}{=} \mathsf{resolve}\ (\Delta^2\ x\ y\ B\ n) \\
&\qquad (\lambda z.\ \mathsf{resolve}^2\ (\mathsf{from\_signal}\ x\ (\mathsf{SUC}\ n))(\mathsf{from\_signal}\ y\ (\mathsf{SUC}\ n))\ B)\ \bot
\end{aligned}$$

Both operators Δ and Δ^2 yield functions of natural argument satisfying the signal condition, i. e. $\vdash \forall x\ A.\,Is\_signal(\Delta\ x\ A)$ and $\vdash \forall x\,y\,B.\ Is\_signal(\Delta^2\,x\,y\,B)$ hold and hence two signal transformers APPLY and APPLY^2 can be defined:

$$\mathsf{APPLY}\ A\ x \overset{\text{def}}{=} \mathsf{to\_signal}(\Delta\ x\ A) \quad \mathsf{APPLY}^2\ B\ x\ y \overset{\text{def}}{=} \mathsf{to\_signal}(\Delta^2\ x\ y\ B).$$

The functional equivalence of S-structures introduced below does not take into account the time when signal values are emitted or received. It ignores the names and the values of the intermediate signals as well. The functional semantics $[\![P]\!]$ of an S-structure P is a boolean higher order function. Its input and output parameters are (abstractions of) signal values of the type $\alpha\ \mathsf{signal}$.

$$\|\ \mathsf{P\_proc}(P : \alpha \rightarrow \beta\,\mathsf{partial})\ \|\ (x : \alpha\,\mathsf{signal}, y : \beta\,\mathsf{signal}) \overset{\text{def}}{=} (y = \mathsf{APPLY}\ P\ x)$$

$$\| \mathsf{Dfg\_proc}\, f \parallel (x : \alpha\,\mathsf{signal}, y : \beta\,\mathsf{signal}) \stackrel{\text{def}}{=} (y = \mathsf{APPLY}\,(\mathsf{Def} \circ f)\, x)$$

$$\| \mathsf{Double} \parallel (x : \alpha\,\mathsf{signal}, y_1 : \alpha\,\mathsf{signal}, y_2 : \alpha\,\mathsf{signal}) \stackrel{\text{def}}{=} (y_1 = x) \wedge (y_2 = x)$$

$$\| \mathsf{Split} \parallel (x : (\alpha \times \beta)\,\mathsf{signal}, y_1, y_2) \stackrel{\text{def}}{=}$$
$$(y_1 = \mathsf{APPLY}\,(\mathsf{Def} \circ \mathsf{FST})\, x) \wedge (y_2 = \mathsf{APPLY}\,(\mathsf{Def} \circ \mathsf{SND})\, x)$$

$$\| \mathsf{Join} \parallel (x_1 : \alpha\,\mathsf{signal}, x_2 : \beta\,\mathsf{signal}, y) \stackrel{\text{def}}{=} (y = \mathsf{APPLY}^2\ \mathsf{Def}\ x_1\, x_2)$$

$$\| \mathsf{Synchronize} \parallel (x_1 : \alpha\,\mathsf{signal}, x_2 : \beta\,\mathsf{signal}, y_1 : \alpha\,\mathsf{signal}, y_2 : \beta\,\mathsf{signal}) \stackrel{\text{def}}{=}$$
$$(y_1 = \mathsf{APPLY}^2\,(\mathsf{Def} \circ \mathsf{FST})\, x_1\, x_2) \wedge (y_2 = \mathsf{APPLY}^2\,(\mathsf{Def} \circ \mathsf{SND})\, x_1\, x_2)$$

To define the functional semantics of the K-process Fork we have introduced an auxiliary function $ForkLen$. $ForkLen(x : (\alpha \times \mathsf{bool})\,\mathsf{signal})n$ delivers a pair (l_1, l_2) of natural numbers where $l_1(l_2)$ is the number of truth values F (resp.T) in the sequence $\mathsf{SND}(\mathsf{from\_signal}\ x\ 0), \ldots, \mathsf{SND}(\mathsf{from\_signal}\ x\ n)$.

$$\| \mathsf{Fork} \parallel (x : (\alpha \times \mathsf{bool})\,\mathsf{signal}, y_F : \alpha\,\mathsf{signal}, y_T : \alpha\,\mathsf{signal}) \stackrel{\text{def}}{=}$$
$$(\forall n.\ \mathsf{let}\ (l_F, l_T) = ForkLen\, x\, n\ \mathsf{in}\ \mathsf{resolve}\ (\mathsf{from\_signal}\, x\, n)\ (\mathsf{Def} \circ \mathsf{FST} =$$
$$(\lambda z.\ \mathsf{if}\ \mathsf{SND}\ z\ \mathsf{then}\ \mathsf{from\_signal}\ y_T(l_T - 1)\ \mathsf{else}\ \mathsf{from\_signal}\ y_F(l_F - 1)))$$
$$(\mathsf{Idle}(\mathsf{from\_signal}\ y_F)l_F) \wedge (\mathsf{Idle}(\mathsf{from\_signal}\ y_T)l_T)) \wedge$$
$$\mathsf{let}\ \emptyset = (\lambda n.\ 0)\ \mathsf{in}\ ((\mathsf{FST} \circ (ForkLen\ x) = \emptyset) \Rightarrow (y_F = \mathsf{blank})) \wedge$$
$$((\mathsf{SND} \circ (ForkLen\ x) = \emptyset) \Rightarrow (y_T = \mathsf{blank}))$$

To define the functional semantics of the K-process Choose we have introduced two auxiliary functions $ChoiceLen$ and $Choice$. A call $ChoiceLen\, n\, x\, y$ delivers a triple $(nextready, l_1, l_2)$, where $nextready = \mathsf{T}$ iff the first input channel is ready after $n + 1$ firings of Choose on input channels x and y, and l_1 and l_2 are the numbers of values read from x and y, resp., after these $n + 1$ firings.

$$Choice\ (0 : \mathsf{num})\ (x : (\alpha \times \mathsf{bool})\,\mathsf{signal})\ (y : (\alpha \times \mathsf{bool})\,\mathsf{signal})\ (z : \alpha\,\mathsf{signal}) \stackrel{\text{def}}{=}$$
$$\mathsf{resolve}\,(\mathsf{from\_signal}\, z\, 0)$$
$$(\lambda z.\ \mathsf{resolve}\,(\mathsf{from\_signal}\, x\, 0)(\lambda p.\ z = \mathsf{FST}\ p)\ \mathsf{F})(x = \mathsf{blank})$$

$$Choice\ (\mathsf{SUC}\ n)\ x\ y\ z \stackrel{\text{def}}{=} \mathsf{let}\ (nextready, l_1, l_2) = ChoiceLen\ n\ x\ y\ \mathsf{in}$$
$$\mathsf{let}\ s = \mathsf{if}\ nextready\ \mathsf{then}\ \mathsf{from\_signal}\ x\ l_1\ \mathsf{else}\ \mathsf{from\_signal}\ y\ l_2\ \mathsf{in}$$
$$\mathsf{resolve}\,(\mathsf{from\_signal}\ z\ (\mathsf{SUC}\ n))\ (\lambda z.\ \mathsf{resolve}\ s\ (\lambda p.\ (z = \mathsf{FST}\ p))\ \mathsf{F})$$
$$(\mathsf{Idle}(\mathsf{from\_signal}\ x)l_1) \wedge (\mathsf{Idle}(\mathsf{from\_signal}\ y)l_2)$$

$$\| \mathsf{Choose} \parallel (x : (\alpha \times \mathsf{bool})\,\mathsf{signal}, y : (\alpha \times \mathsf{bool})\,\mathsf{signal}, out : \alpha\ \mathsf{signal}) \stackrel{\text{def}}{=}$$
$$(\forall n.\ Choice\ n\ x\ y\ out) \wedge ((\forall n.\ \mathsf{FST}(ChoiceLen\ n\ x\ y)) \Rightarrow (y = \mathsf{blank}))$$

$$\| \mathsf{Source}\ (C : \alpha) \parallel (out : \alpha\ \mathsf{signal}) \stackrel{\text{def}}{=} (\forall n.\ (\mathsf{from\_signal}\ out\ n = \mathsf{Def}\ C)) \vee$$
$$(\exists m.\ \forall n.\ (\mathsf{from\_signal}\ out\ n = \mathsf{if}\ n < m\ \mathsf{then}\ \mathsf{Def}\ C\ \mathsf{else}\ \bot))$$

$$\| \mathsf{Sink} \parallel (in : \alpha\ \mathsf{signal}) \stackrel{\text{def}}{=} \mathsf{T}$$

The functional semantics of an S-structure $\mathcal{S} = \exists\, \bar{b}\,.\, S_1 \wedge \ldots \wedge S_n$ is defined by $\parallel \mathcal{S} \parallel \overset{\text{def}}{=} \exists\, \bar{b}.\ \parallel S_1 \parallel \wedge \ldots \wedge \parallel S_n \parallel$. We call two S-structures S_1 and S_2 *functional equivalent*, if their functional semantics are equal: $S_1 \approx S_2 \overset{\text{def}}{=}\ \parallel S_1 \parallel = \parallel S_2 \parallel$.

6 Program Transformations

In what follows we present some functional equivalence theorems which can be considered as Gropius program transformations and build the basis for circuit combining/partitioning algorithms. All of the theorems have been proved in the theorem prover HOL [4].

$$\begin{gathered}\vdash\ \exists\,(r : (\alpha \times \mathsf{bool})\,\mathsf{signal})(s : \alpha\,\mathsf{signal}).\ \mathsf{Dfg\_proc}\,(\lambda x.\,(x, \mathsf{T}))(in, r)\ \wedge \\ \mathsf{Fork}\,(r, s, out) \wedge \mathsf{Sink}(s) \\ \approx \\ \mathsf{Dfg\_proc}\,(\lambda x.\ x)(in : \alpha\ \mathsf{signal}, out : \alpha\ \mathsf{signal})\end{gathered} \tag{1}$$

In transformation (1) we took advantage of the fact that channel r can contain only sequences of the signal values $\mathsf{Def}(x, \mathsf{T})$ or $\bot$.

$$\begin{gathered}\vdash\ \exists\,(r : \alpha\,\mathsf{signal}).\ \mathsf{Source}\,C\ (r)\ \wedge \mathsf{Join}\,(r, in : \beta\,\mathsf{signal}, out) \\ \approx \\ \mathsf{Dfg\_proc}\,(\lambda x : \beta.\,(C, x))(in, out : (\alpha \times \beta)\,\mathsf{signal})\end{gathered} \tag{2}$$

We use the transformation (2) for elimination of K-processes Source by constant propagation.

$$\begin{gathered}\vdash \exists\, r\ s.\ \mathsf{Double}\,(x : \alpha\,\mathsf{signal}, r, s) \wedge \mathsf{P\_proc}\ A\,(r, u) \wedge \mathsf{P\_proc}\ A\,(s, v) \\ \approx \\ \exists\,(t : \beta\,\mathsf{signal}).\ \mathsf{P\_proc}\,(A : \alpha \to \beta\ \mathsf{partial})(x, t) \wedge \mathsf{Double}\,(t, u, v)\end{gathered} \tag{3}$$

By the application of theorem (3) the double execution of a P-process is avoided: two copies are combined into a single one. Theorem (4) allows to combine two P-processes, which are executed in sequence, into a single P-process, or to partition a given P-process into two successively executed P-processes.

$$\vdash\ (\exists\, u.\, \mathsf{P\_proc}\ A\ (x, u) \wedge\ \mathsf{P\_proc}\ B\ (u, y)) \approx (\mathsf{P\_proc}\ (A\ \mathsf{SERIAL}\ B)\ (x, y)) \tag{4}$$

Theorem (5) describes the functional equivalence of an S-structure consisting of two parallel P-processes, whose results are combined over a K-process Join, to a structure where first the two inputs are combined using Join into a single value which is then forwarded to a single P-process. With the help of this theorem a P-process can be partitioned into two parallel ones or two different P-processes can be combined to a single one.

$$\vdash \quad \begin{array}{c} (\exists\, u\ v.\ \mathsf{P\_proc}\, A\ (x,u) \wedge \mathsf{P\_proc}\, B\ (y,v) \wedge \mathsf{Join}(u,v,z)) \\ \approx \\ (\exists\, w.\ \mathsf{Join}(x,y,w) \wedge \mathsf{P\_proc}\ ((A \circ \mathsf{FST})\ \mathsf{PARALLEL}\ (B \circ \mathsf{SND}))(w,z)) \end{array} \tag{5}$$

7 Conclusion

We have presented a method for the formal synthesis at the system level. In our approach, systems are described as structures of concurrent processes. The synthesis of correct circuit implementations is performed by means of equivalent transformations, which correspond to the application of theorems in the theorem prover HOL. For the proof of the correctness of transformations, the functional equivalence relation on process structures has been introduced and theorems about correctness of some optimising transformations have been proved. While the behavioral equivalence of S-structures introduced in [3] takes the exact temporal behavior of the circuit into account, the functional equivalence considers only the functionalities of the underlying processes.

References

1. C. Blumenröhr and D. Eisenbiegler. *Performing High-Level Synthesis via Program Transformations within a Theorem Prover*, In: Digital System Design Workshop at the 24th EUROMICRO 98 Conference, 1998.
2. C. Blumenröhr and V. Sabelfeld. *Formal Synthesis at the Algorithmic Level*, In: Correct Hardware Design and Verification Methods, Charme'99, 1999.
3. C. Blumenröhr. *A formal approach to specify and synthesize at the system level*, In: M. Mutz and N. Lange, eds., GI/ITG/GME Workshop Modellierung und Verifikation von Schaltungen und Systemen, Braunschweig, Germany, February 1999, pp. 11-20, Shaker-Verlag.
4. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
5. J. Iyoda, A. Sampaio, L. Silva. *ParTS: A Partitioning Transformation System*, In: FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, LNCS 1709, pp. 1400–1419.
6. K. Jensen. *Colored Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Vol. 1: Basic Concepts, Springer Verlag 1992.
7. M. Larsson. *An engineering approach to formal digital system design.* The Computer Journal, 38(2):101–110, 1995.
8. T.F. Melham. *Automating Recursive Type Definitions in Higher Order Logic*, In: G. Birtwistle and P. A. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, pp. 341-386, Springer-Verlag, 1989.
9. R. Sharp, O. Rasmussen. *The T-Ruby design system.* In: IFIP Conference on Hardware Description Languages and their Applications, pp. 587–596, 1995.

Automated Program Synthesis for Java Programming Language*

Mait Harf, Kristiina Kindel, Vahur Kotkas, Peep Küngas, and Enn Tyugu

Department of Computer Science,
Institute of Cybernetics at Tallinn Technical University, Estonia
{mait,kristina,vahur,peep,tyugu}@cs.ioc.ee

Abstract. In this paper we introduce a methodology of program synthesis for Java programming language by extending Java classes with high level specifications. The specifications are handled by a distributed synthesizer also briefly described in this paper.

1 Introduction

The number of ready to use software components grows and software designers are faced with the fact that soon they are unable to have an overview of all software libraries. To solve this problem, we should use automated software design, where software libraries are handled automatically with the help of software specifications provided by a designer.

One way to automate the software design process is to use Structural Synthesis of Programs (SSP) [1]. SSP is a technique of deductive synthesis of programs based on automatic proof search in intuitionistic propositional calculus. The solving complexity is hidden from the end-user into the system. The idea of using SSP for automated program generation is not new. Already in seventies a Priz family of programming languages was developed in the Institute of Cybernetics, that allows engineers to solve their tasks using a very high-level programming language. A similar approach has justified itself quite well in the Amphion system [2].

In this paper we introduce a methodology of extending Java programming language with capabilities of SSP. This methodology helps us to cut down the programming time and expenses, and decreases the amount of possible faults as the actual software is generated automatically. We aim to create a new tool that adds to the Java language declarative specifications that are used for automated program construction. Our ultimate goal is to increase the programming efficiency through the use of general solvers for variety of problems and software reuse. So we move towards the future's realm, where a programmer needs only to specify what the program should perform, not how it should do it.

* This work was partially funded by Estonian Innovation Foundation under the contract No. 6kl/00.

The Java language was chosen to be the platform for the SSP addition because of its relative robustness and flexibility. Combining it with CORBA technology [3] provides us with more flexible and concurrent computing in a network and forces to follow the ideas of modular programming.

When applying the SSP to Java we have the following in mind:

- Prototyping — no need to implement the (efficiently working) algorithms in the early phase of program development, as the system would take care of it automatically.
- Software reuse — by applying the declarative specifications to the existing Java classes we can automate their reuse by letting the system handle the search for suitable components for the software.

In Sect. 2 we give a general overview of the extended specification language, using an example to highlight its main features. The architecture to carry out the automated program construction process using distributed object model is proposed in Sect. 3. In Sect. 4 we briefly discuss the working principles of the Planner that is the heart of the automated program construction.

This work is inspired by the work done in the Institute of Cybernetics, Estonia during several decades and related also to the work of Sven Lämmerman [4] from Royal Institute of Technology, Sweden.

2 An Example and the Specification Language

In the current section we will use a modeling of radar coverage as an example. When starting to model something, we usually take a handbook of the field of interest and study it. When the area of interest is radar performance calculation the main equation that can be found is:

$$R^4 = \frac{P_{\mathrm{t}} G^2 \lambda^2 \sigma F}{(4\pi)^3 SNR_{\min} LN}, \tag{1}$$

where P_{t} is transmit power, G is antenna gain, λ is wavelength, σ is target radar cross section, F is pattern propagation factor, $SNR_{\min}$ is minimal signal to noise ratio for target detection with certain probabilities, L is signal losses, N is total received noise and R is detection range. It is obvious to include the same components into our model. So we create a new Java class called *Radar* and declare the listed components.

We add a declarative specification to the Java class that describes the relations (constraints) among the components. The declarative specification is added to a Java class as a string array *SSPspec*, where every string represents a single declaration, also called as clause.

In the specification we have to redefine all the components (Java variables and objects) of the class (statements starting with *var* or *vir*), that would be used in the relations (statements starting with keyword *rel*). A sample class of a radar, extended with a structural specification, is shown on Fig. 1.

```
public class Radar implements SSPinterface {
    public static String[] SSPspec = {
       "var r, wavelen : any", // detection range, wavelength
       "var pt, prf, rcst : any", // power, pulse rep. freq., target size
       "var g : any", // antenna power gain
       "var ppf : any", // Pattern Propagation Factor
       "var snr_min : any", // minimal Signal to Noise Ratio
       "var losses : any", // system losses
       "var noise : any", // total noise power
       "rel r^4 = pt * g^2 * wavelen^2 * rcst * ppf "+
              " / ((4*pi)^3 * snr_min * losses * noise))",
       "rel snr_min.prf == prf",
       "rel noise.r == r",
       "rel noise.prf == prf",
    };
    Length r, wavelen;
    Integer pt, prf, rcst;
    Gain g;
    Noise noise;
    Losses losses;
    Ratio snr_min;
    Pattern ppf;
}
```

Fig. 1. Class Radar

Note that the components specified as of type **any** have to be actual components of the Java class and their type is determined from class declarations.

In the class *Radar* there are two types of relations present: equation and equivalences. Two components are equivalent when their evaluation is always the same during a computation. Our system includes an equation solver, hence we can use equations as relations in the specification.

The first component r is of class *Length* and denotes the range from radar to target. The class *Length* is represented in Fig. 2. In the SI (System International) length is measured in meter (m). The class *Length* supports automatic transformation of units (length measured in other units can be automatically translated into meters.). By default the length is in meters (*var* component of the class *Length*).

There is a special usage for having only one *var* component in a class: in such case we can use this object in relations without specifying its component to be used. The *var* component is automatically substituted instead of the whole object. This is used for example in the case of components of class *Length* in the equivalence specification of class *Radar*. There we write `"rel noise.r == r"` instead of `"rel noise.r.m == r.m"`.

The relations specify the conversion rules among the components. In the class *Length* all the relations are equations that can be treated as two-way translation rules. For example clause `rel m/1000 = km` can be treated as two methods. One

```
public class Length implements SSPinterface {
    public static String[] SSPspec = {
        "var m : any", // main measurement unit in SI
        "vir km, nmi, cm : any", // other units
        "rel m / 1000 = km",
        "rel m / 1852 = nmi",
        "rel m = cm / 100",
    };
    Float m, km, nmi, cm;
    public void Length(Float number, String component) {
        if (component.equals("m")) m = new Float(number);
        else if (component.equals("km")) km = new Float(number);
        else if (component.equals("nmi")) nmi = new Float(number);
        else if (component.equals("cm")) cm = new Float(number);
    }
}
```

Fig. 2. Class Length

allows to calculate m from km and another km from m. In the similar manner we can define classes for other components of class *Radar*.

In order to use the specifications we have to include a component of the class *Radar* in another class (let us call it *RadarCoverage* (see Fig. 3)). Now we have to evaluate some of it's components (see the constructor in class *RadarCoverage*) and invoke a method *compute* of class SSP. A synthesizer (see Sect. 3) is then executed. It searches for a way of computations to find values for all *var* components of that object unless we explicitly specify what components should be calculated. The synthesizer uses the declarative specification given in classes that implement the $SSPinterface$.

Similar methodology has been used in a radar coverage modeling package in a programming environment NUT [5]. Experiments show that the methodology is suitable for solving this kind of problems.

3 The Distributed Synthesizer

Composing a distributed application — an application composed of distributed components — adds the following features:

- Higher flexibility — as the components are not compiled into the application, the changes in the components do not affect the consistency of the distributed application if the interfaces of these components are fixed and semantics of their inputs and outputs remains unchanged.
- Higher fault-tolerance — if a distributed component is developed for a certain environment and is well tested to work properly in it, then composing an application using these distributed components, that reside always in their native environment, cuts down in programming and testing time.

```
public class RadarCoverage implements SSPinterface {
    public static String[] SSPspec = {
        "var radar : any",
        "vir ver_cov : any",
        "rel [radar.hor_ang, radar.ver_ang -> radar.r]"
                                         + "-> ver_cov {CalcVerCov}"
    };
    Radar radar;
    Coord[] ver_cov;
    public void RadarCoverage() { //The constructor
        radar = new Radar();
        radar.wavelen = new Length(3.25, "cm");
    // ..... other initializations in the same way ......
    }
    public static void main(String[] args) {
        RadarCoverage model = new RadarCoverage();
        SSP.compute(model, "ver_cov"); //Invoking the synthesizer
    }
    public Coord[] CalcVerCov() { //Calculates vertical coverage diagram
        //method body
    }
}
```

Fig. 3. Class RadarCoverage

In a distributed application the intercomponent communication is fairly expensive in terms of time and other resources [6]. Thus components are encouraged to be larger. We propose an architecture of the synthesizer, which is decomposed into 6 logically separated components: Knowledge Base (KB), Compiler, Decorator, Planner, Code Generator and Component Repository. The interconnections between these components are presented in Fig. 4.

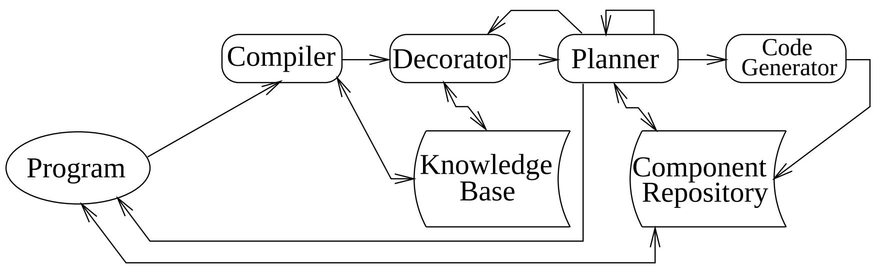


Fig. 4. Architecture of the synthesizer

In the class *RadarCoverage* we have a relation that specifies how to calculate a radar vertical coverage. For vertical coverage calculation a method

CalcVerCov can be used as soon as a subtask that calculates component r of *radar* from given vertical and horizontal angle and other components that were evaluated before, is solvable. The method forms a vertical coverage diagram using this subtask in a loop for calculating the maximal detection ranges for several elevation angles.

The task of the Compiler is to parse declarative specifications and store the results into a KB. The KB represents a memory system designed to hold data structures being used later for planning structure creation. It manages SSP specifications.

The Decorator creates a special structure suitable for fast planning, sets evaluation states of components on the created structure and passes it all to the Planner.

The Planner (see Sect. 4) is used for automatic program synthesis from problem description. As a result of planning we get an algorithm that is not necessarily the shortest, however it does not contain unnecessary relations.

If the solution is found, class code is generated by the Code Generator. The code is then compiled to Java class and added to the Component Repository.

The aim of Component Repository is to maintain a set of components solving a certain problem. The components can be reused to solve similar tasks when the problem description is matching.

The Compiler, the Decorator, the Planner and the Code Generator access the KB to set and get information needed during specification processing.

4 The Planner

The Planner uses SSP and the proof search algorithms proposed in [7].

The declarative specification is located in the KB — a special structure to store descriptions of all objects (components of the context where planner was invoked) and all relations (computability statements), describing dependencies between objects. Two kind of relations may occur in the declarative specification:

- Unconditional relations implementing unconditional computability statements of SSP. In such relations computability of some (output) objects depends only on some other (input) objects. Unconditional relations of several types as equations, equivalences, Java methods etc. are available in the specification language.
- Relations with subtasks implementing conditional computability statements of SSP. Such relations describe more sophisticated dependencies where output objects depend not only on input objects but also on solvability of some other computing problems.

The problem specification is of form $x \longrightarrow y$, where x denotes the set of known objects and y denotes the set of objects to be computed. The Planner has to construct an algorithm (a sequence of relations) that describes how to compute y from x.

The proof search strategy of SSP applied in the Planner is:

- an assumption-driven forward search to select unconditional relations (linear planning). The algorithm works in the forward direction and only unconditional relations are considered. At starting point all the objects that are inputs for the current problem are set as known objects. At each step, if all the input objects of the relation are known and at least one output object in not known, the relation is added to the synthesizable algorithm. After using relation to the synthesizable algorithm all its output objects are set as known objects. The search is a simple flow analysis on the network of unconditional relations.
- a goal-driven backward search to select and solve subtasks. The search is applied if the assumption-driven forward search is not sufficient. Only such relations with subtasks are considered which input objects are known. The Planner is recursively used for solving every subtask of the relation. If all the subtasks of the relation are solved, the relation is added to the synthesizable algorithm. Linear planning is used after every invocation of a relation with subtasks in the algorithm.
- a minimization is applied to the synthesized algorithm. Using search strategies mentioned above does not guarantee that we have built the shortest possible algorithm for computing the desired goal. Even more, the synthesized algorithm may contain relations that are not necessary for computing the goal. Minimization is used to exclude such relations from synthesized algorithm. As a result of planning we get an algorithm that is not necessarily the shortest, but it does not contain unnecessary relations.

5 Concluding Remarks

In the current paper we propose an extension to the Java language that increases the programming efficiency through the use of general solvers for variety of problems and software reuse.

We also propose an architecture for the synthesizer, that performs the automated program construction.

Our main objective is to extend a distributed programming language with SSP capabilities that automates software reuse and helps users to design programs. We intend to create a tool that allows engineering modeling, prototyping, flexible networking, and connections among different software components designed and written for different platforms.

We use a distributed object model that increases the flexibility of particular system and provides higher flexibility and fault-tolerance.

References

1. E. Tyugu. The structural synthesis of programs, Lecture Notes in Computer Sciences, Vol. 122, 1981, pp. 290–303.

2. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood. Deductive Composition of Astronomical Software from Suroutine Libraries. In: 12th Conference on Automated Deduction. A. Bundy, (ed). Springer-Verlag Lecture Notes in Computer Science, Vol. 814, 1994.
3. S. Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine, Vol. 14, No. 2, February 1997.
4. S. Lämmermann. Automated Composition of Java Software, thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, May 2000.
5. V. Kotkas, J. Penjam. Ontology-based design of surveillance systems with NUT. Proceedings of the Third International Conference on Information Fusion, Paris, Vol. 2, 2000.
6. D. Budgen, P. Brereton. Component-Based Systems: A Classification of Issues. Computer (IEEE CS), November 2000, Vol. 33, No. 11, pp. 54–62.
7. M. Harf, E. Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, Vol. 6, 1980, pp. 165–175.

The Varieties of Programming Language Semantics

And Their Uses

Peter D. Mosses

BRICS & Department of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
`pdmosses@brics.dk` `http://www.brics.dk/~pdm`

Abstract. Formal descriptions of syntax are quite popular: regular and context-free grammars have become accepted as useful for documenting the syntax of programming languages, as well as for generating efficient parsers; attribute grammars allow parsing to be linked with type-checking and code generation; and regular expressions are extensively used for searching and transforming text. In contrast, formal semantic descriptions are widely regarded as being of interest only to theoreticians. This paper surveys the main frameworks available for describing the dynamic semantics of programming languages. It assesses the potential and actual uses of semantic descriptions, and considers practical aspects, such as comprehensibility, modularity, and extensibility, which are especially significant when describing full-scale languages. It concludes by suggesting that the provision of mature tools for transforming practical semantic descriptions into reasonably efficient compilers and interpreters would significantly increase the popularity of formal semantics.
The paper is intended to be accessible to all computer scientists. Familiarity with the details of particular semantic frameworks is not required, although some understanding of the general concepts of formal semantics is assumed.

1 Introduction

▷ Formal *syntax* is widely used in practical applications.

Formalisms for specifying the syntax of programming languages originated from theoretical studies of automata and parsing. They have subsequently been found useful in various practical applications. For instance, tools such as `lex` and `yacc` generate scanners and parsers from regular, resp. LALR(1) context-free grammars; the ASF+SDF Meta-environment [14] generates efficient parsers from unrestricted context-free grammars [8]; compiler-writing systems such as Eli [30] generate type-checkers and code-generators from attribute grammars; and editors such as `vi` and `emacs` support the use of regular expressions to search efficiently for (and replace) particular patterns in text.

▷ Formal *semantics* is seldom used in practical applications.

In marked contrast to the popularity of formal syntax, formal semantic descriptions have seldom been exploited in practical applications concerning design and implementation of programming languages. As we shall see in Sect. 2, there is no shortage of semantic frameworks to choose from, nor has there been a lack of theoretical effort in establishing the foundations of the various frameworks. The major semantic frameworks have also been quite widely taught (even at the undergraduate level) and plenty of pedagogical text-books are available. The potential benefits of using formal semantics in general, as well as the special advantages of particular frameworks, have been proclaimed in numerous books and papers. Yet despite all this investment of effort in promoting formal semantics, significant practical uses of semantic descriptions have been rather few and far between.

▷ We shall survey the various semantic frameworks, list some uses that have been made of them, and speculate on the hindrances for greater use.

Our survey of semantic frameworks in Sect. 2 will be deliberately brief and superficial, focussing on the most important differences between the frameworks, and ignoring many details. In Sect. 3 we list the potential uses of formal semantics, and indicate some of the more interesting actual uses. Section 4 considers various hindrances to greater use of semantic descriptions.

▷ Our conclusion will be that wider use of formal semantics depends on the availability of tools for generating (reasonably efficient) implementations from semantic descriptions.

Some frameworks may appear to be inherently better-suited for generating efficient implementations. For instance, it might be imagined that operational semantics would have decisive advantages over the more abstract denotational and axiomatic approaches. However, any kind of semantics is supposed to *determine* the observable behaviour of all programs in the described language; provided that the relevant information about behaviour can be extracted automatically from the semantics, the generation of implementations from that information may be largely independent of how it was originally presented. In general, the efficiency of the generation process itself may well be unimportant compared to the efficiency of running programs via the generated implementation.

▷ Few such tools are currently available.

Currently, very few systems generating *any* kind of implementation from semantic descriptions are available [25]. Most of the semantics-directed compiler and interpreter generators reported in the literature were developed in the 1980's, and remained as prototypes with limited support and maintenance.

▷ Few semantic descriptions have good pragmatic features.

The use of even a properly-implemented and well-maintained system is limited to the input available for it. Here, the input should be complete, fully formal semantic descriptions—in marked contrast to the kind of semantic description usually found in textbooks and papers, where “obvious” details are often omitted, and semi-formal “conventions” are introduced in the interests of conciseness. Good pragmatic features, such as modularity and readability, are needed for efficient development and use of semantic descriptions, but are sadly lacking in most frameworks.

▷ The main points of this paper are all displayed like this.

For a quick first reading, one may prefer to focus on the main points and illustrative examples, skipping most of the intervening text. It is hoped that the display of the main points (following Alexander [3]) does not significantly hinder a continuous reading of the entire text.

2 Formal Semantics

To start with, let us distinguish between *static* and *dynamic* semantics:

▷ *Static semantics* concerns checking for *well-formedness.*

For languages intended to be implemented by compilers, the well-formedness of every part of a program should be checked before starting to run the program, in general. The static semantics of a program corresponds to its compile-time behaviour, and is independent of any input that is provided to the program at run-time. Well-formedness is usually decidable, so static semantics may be treated as a kind of (context-sensitive) syntax, and specified by attribute grammars.

▷ *Dynamic* semantics is about *computation.*

For compiled programs, their well-formedness has already been checked, and their dynamic semantics corresponds just to their run-time behaviour. For languages implemented by interpreters, programs are usually run without a foregoing overall well-formedness check, and any required checks happen at run-time, thus being considered part of dynamic semantics.

▷ In this paper, the focus is entirely on *dynamic* semantics.

Static semantics is usually an essential ingredient in complete language descriptions, and an interesting topic in its own right. Unfortunately, it is outside the scope of the present paper. Note however that static semantics (especially for type-checking and type-inference) appears to be used more often in practical applications than dynamic semantics is.

▷ Most frameworks for dynamic semantics can be classified as *operational*, *denotational*, or *axiomatic*.

In operational frameworks, the semantics of a program is specified as an abstract machine or transition system, the computations of which represent possible executions of the program. A denotational semantics is given by inductively-defined functions mapping each program to an abstract entity representing its observable behaviour, and each part of a program to an abstract entity representing its contribution to that behaviour. Axiomatic semantics involves rules for deducing assertions about the correctness or equivalence of programs and their parts.

We survey the main frameworks that have been developed within each of the above classifications, and briefly consider a proposal to develop complementary descriptions. A few frameworks do not fall into these classifications, being essentially *hybrids* of different kinds of frameworks.

▷ Most semantic frameworks are based on *context-free abstract syntax*.

Use of *abstract* syntax implies that semantics is defined for *trees* that directly reflect the actual compositional structure of programs. Concrete syntax, with the issue of how to parse a program text so as to discover its structure, is thus of no concern in semantic descriptions. Of course, a complete language description has to specify the exact relationship between concrete and abstract syntax—which may sometimes be not so straightforward, especially when concrete syntax has been specified with a restricted kind of grammar, such as LALR(1). *Context-free* abstract syntax is particularly pleasant to work with, and has clean and simple algebraic foundations.

In most semantic frameworks, abstract syntax is specified by ordinary BNF-like grammars—the terminal symbols being as in concrete syntax, but here used merely to distinguish between the various abstract constructs. Using the same terminal symbols makes it easy to guess the intended relationship between concrete and abstract syntax, and generally facilitates the reading of semantic descriptions. However, such abstract syntax grammars are always interpreted as defining sets of *trees*, rather than sets of strings. These grammars tend to be significantly simpler than the corresponding unambiguous context-free grammars for concrete syntax—in particular, they may have significantly fewer nonterminal symbols and chain-productions.

▷ We shall illustrate the various semantic frameworks with fragments involving the description of a simple if-statement and an assignment expression.

An abstract syntax for if-statements, expression-statements, and assignment expressions (as may be found in languages such as C and Java) is specified in Table 1. We do not try to make a syntactic distinction between boolean and other expressions, since the types of variables in expressions usually depend on their declarations, making the syntax of well-typed expressions context-sensitive.

Table 1. Abstract syntax

$$\begin{array}{l} s \in \mathit{Stm} ::= \texttt{if(}\ \mathit{Exp}\texttt{)}\,\mathit{Stm} \mid \mathit{Exp}\ \texttt{;} \mid \ldots \\ e \in \mathit{Exp} ::= \mathit{Var}\,\texttt{=}\,\mathit{Exp} \mid \ldots \\ x \in \mathit{Var} \end{array}$$

When formulating dynamic semantics, we need not worry about what semantics is given to programs containing ill-typed expressions, assuming that such programs are filtered out by a preceding static semantics.

The grammar in Table 1 is unambiguous, but in fact with grammars for abstract syntax, ambiguity does not give rise to any problems. For instance, a production such as $\mathit{Exp} ::= \mathit{Exp}\,\texttt{==}\,\mathit{Exp}$ simply specifies that both the left and right sub-trees of the trees for this construct are of the same sort, namely *Exp*. Incidentally, some authors prefer to use ordinary variables, rather than sort names, as nonterminal symbols in abstract syntax grammars, writing for example $e ::= e_1\,\texttt{==}\,e_2$ instead of $\mathit{Exp} ::= \mathit{Exp}\,\texttt{==}\,\mathit{Exp}$.

▷ Formal semantics aims at *modelling* the observable behaviour of complete programs.

In the models used in formal semantics, low-level implementation-dependent details are generally ignored, e.g. bounds on the size of numbers or arrays, limits on the depth of recursive procedure calls, or the way in which memory is allocated and freed. The actual binary representation of values is ignored too, and computed values are modelled as abstract mathematic entities. For instance, the values computed by our illustrative expressions $e \in \mathit{Exp}$ may be modelled as abstract values $v \in V$, including the familiar mathematical integers $n \in Z$ and the boolean values $\mathit{tt}, \mathit{ff} \in B$.

As usual when giving a model, auxiliary entities may be introduced purely to facilitate the construction of the model. For dynamic semantics, the auxiliary entities typically include environments $\rho \in \mathit{Env} = \mathit{Ide} \to L$, stores $\sigma \in S = L \to V$, and locations $l \in L$. The auxiliary entities do not themselves have to correspond to actual components of implementations.

Many frameworks require semantics to be specified not only for complete programs, but also for all their parts (statements, declarations, expressions, etc.). The semantics of such parts are regarded as auxiliary entities too, since implementations are not obliged to follow the compositional structure of the language. For our purposes here, however, we shall not bother to distinguish complete programs from other syntactic constructs.

▷ We focus on frameworks of current interest.

Our purpose here is not to give a comprehensive historical account of the development of the various frameworks for formal semantics, but rather to limit our attention to frameworks of current interest.

2.1 Operational Semantics

Various frameworks for operational semantics of programming languages have been developed, starting from the early 1960's [33]. Here, we consider Structural Operational Semantics and Natural Semantics (together with their recently-developed modular variants), Reduction Semantics, and the Abstract State Machine approach.

▷ *Operational* semantics models the *computations* of programs.

A computation is usually regarded as a (perhaps infinite) *sequence of steps* between states. An alternative approach is to represent (terminating) computations as *derivation trees*, where the steps occur as leaves but without explicit sequencing.

▷ The semantics of a program is determined by the set of possible computations, modulo some equivalence relation.

States in computations generally incorporate the abstract syntax of the entire program—or at least, the part of it that remains to be executed. Thus no two syntactically-distinct programs can ever have the same (non-empty) sets of possible computations, even when their differences are obviously insignificant. To obtain a reasonable notion of semantic equivalence in operational semantics, some equivalence relation that ignores the syntactic components of states has to be introduced; a popular choice is bisimulation equivalence [35].

Structural Operational Semantics (SOS) was proposed by Plotkin in 1981 [51]. The main aim was to provide a simple and direct method, allowing concise and comprehensible semantic descriptions based on simple mathematics. The basic ideas of SOS have since been presented (on a less ambitious scale) in various textbooks (e.g. [48]), and exploited in numerous papers on concurrency [34]; see also [2].

▷ Computations are modelled as *sequences* of (possibly labelled) *transitions* between states involving syntax, computed values, and auxiliary entities.

The sequences may be finite or infinite. The states themselves are finite mathematical entities, but in contrast to automata theory, the sets of states here are generally infinite.

Characteristic for SOS is that as a computation proceeds, phrases of the program (i.e. branches of the abstract syntax tree) are gradually *replaced* by the values that they have computed. Thus in an initial state the program tree is purely syntactic, in a final state it has been replaced by its computed value, and in between, it is usually a mixture of syntax and computed values. It may also happen that phrases get replaced by different trees, possibly involving auxiliary constructs not present in the original syntax of the language being described.

Auxiliary components of states often include stores such as $\sigma \in S = L \to V$ and environments such as $\rho \in \mathit{Env} = \mathit{Var} \to L$, where L is some set of locations (i.e. addresses in memory). Environments however may also be taken as an extra argument of the transition relation in a so-called relative transition system [51]; they can be avoided altogether by use of syntactic substitution.[1]

▷ Transitions are specified by axioms and inference rules.

In sequential languages, a step for a compound phrase depends on a step for a sub-phrase, whereas in concurrent languages, it may depend on synchronized steps for more than one phrase. Table 2 shows the axioms and inference rules for the constructs whose abstract syntax was specified in Table 1.

Table 2. Structural operational semantics

$e \in \mathit{Exp} ::= \ldots \mid V$
$s \in \mathit{Stm} ::= \ldots \mid ()$

$$\boxed{\rho \vdash s, \sigma \longrightarrow s', \sigma'}$$

$$\frac{\rho \vdash e, \sigma \longrightarrow e', \sigma'}{\rho \vdash \texttt{if(}e\texttt{)}s, \sigma \longrightarrow \texttt{if(}e'\texttt{)}s, \sigma'} \tag{1}$$

$$\rho \vdash \texttt{if(}\mathit{tt}\texttt{)}s, \sigma \longrightarrow s, \sigma \qquad \rho \vdash \texttt{if(}\mathit{ff}\texttt{)}s, \sigma \longrightarrow \sigma \tag{2}$$

$$\boxed{\rho \vdash e, \sigma \longrightarrow e', \sigma'}$$

$$\frac{\rho \vdash e, \sigma \longrightarrow e', \sigma'}{\rho \vdash x \texttt{=} e, \sigma \longrightarrow x \texttt{=} e', \sigma'} \qquad \rho \vdash x \texttt{=} v, \sigma \longrightarrow v, \sigma[v/l] \text{ if } l = \rho(x) \tag{3}$$

Natural Semantics was developed by Kahn and his group at Sophia Antipolis in the mid-1980's [29]. It is sometimes referred to as "big-step" SOS. It can be used together with the pure "small-step" SOS—in fact the big-step style is actually just a (very) special case of the small-step style, involving initial and final states but no intermediate states. Plotkin used the transitive closure of a small-step transition relation to specify that a small step of statement execution depended on an entire expression evaluation [51].

▷ Terminating computations are modelled as evaluation relations between syntax and computed values, possibly involving auxiliary entities.

One drawback of natural semantics is that nonterminating computations are generally ignored.[2] On the other hand, it may be seen as an advantage that

[1] The actual definition of substitution is rather tedious, and usually left to the reader's imagination...
[2] Nontermination can be modelled by use of infinitary logic [6,12].

computed values do not need to be allowed as components of abstract syntax trees in natural semantics, in contrast to SOS.

The usual style is to exhibit the environment as an extra argument to the evaluation relation, as illustrated in Table 3; the resemblance to sequents in Gentzen calculi for Natural Deduction led to the name of the framework.

▷ Evaluations are specified by *axioms* and *inference rules.*

In sequential languages, evaluation of a compound phrase depends on the evaluation of all the involved sub-phrases. However, the rules are not strictly inductive, in general: e.g. a (terminating) evaluation of a loop may depend on another evaluation for the same loop. The description of concurrent languages, or even of interleaved expression evaluation, is problematic in pure natural semantics, because of the lack of intermediate states. A compromise between small- and big-step semantics is evaluation to "committed form" [53].

The need to "thread" effects on stores explicitly through premises of rules when describing conventional imperative languages is a considerable disadvantage of Natural Semantics—so much so that when this framework was used for the Definition of Standard ML [36], a convention was introduced so that the store could actually be left implicit in most rules (the premises being written in the intended order of evaluation of sub-expressions). A similar convention was adopted regarding the propagation of exceptions.

Table 3. Natural semantics

$e \in \mathit{Exp}$
$s \in \mathit{Stm}$

$$\boxed{\rho \vdash s, \sigma \longrightarrow \sigma'}$$

$$\frac{\rho \vdash e, \sigma \longrightarrow tt, \sigma' \quad \rho \vdash s, \sigma' \longrightarrow \sigma''}{\rho \vdash \texttt{if(}e\texttt{)}s, \sigma \longrightarrow \sigma''} \qquad \frac{\rho \vdash e, \sigma \longrightarrow f\!f, \sigma'}{\rho \vdash \texttt{if(}e\texttt{)}s, \sigma \longrightarrow \sigma'} \tag{4}$$

$$\boxed{\rho \vdash e, \sigma \longrightarrow v, \sigma'}$$

$$\frac{\rho \vdash e, \sigma \longrightarrow v, \sigma'}{\rho \vdash x \texttt{=} e, \sigma \longrightarrow v, \sigma'[v/l]} \text{ if } l = \rho(x) \tag{5}$$

Modular Operational Semantics was developed by the present author at the end of the 1990's [44,46], with the aim of improving some of the pragmatic aspects of the conventional SOS and natural semantics frameworks—inspired by the improvements that can be obtained in denotational semantics by the use of monads, see Sect. 2.2.

▷ Modular SOS is a variant of SOS where states are restricted to syntax and computed values, and all auxiliary entities are incorporated in *labels* on transitions.

Labels in modular SOS may include all the auxiliary entities normally employed in the conventional SOS framework, for example environments, (pairs of) stores, and (sequences of) communication signals. The set of labels is generally infinite. It is straightforward to reduce a modular SOS to a conventional SOS, by moving the relevant components of the labels back to their usual places in the states.

▷ Computations require adjacent labels to be *composable*.

Composition of labels, written $\alpha_1\ ;\ \alpha_2$, is usually partial: when labels contain environments, they compose only when the two environments are the same; and when they contain pairs of stores, the second store in the first label has to be the same as the first store in the second label. The presence of communication signals, however, does not affect composability.

In fact the set of labels always forms a *category*. Let the variable α range over all labels, but let ι range only over *identity* labels. The objects of the label category may be regarded as the semantic components of states—which are clearly separated from the syntactic components in this framework, in contrast with all other operational frameworks. Notice in Table 4 how arbitrary labels α are propagated during the evaluation of sub-expressions, whereas labels on reduction steps are required to be identities ι.

New components can be added to labels by applying *label transformers*, which form product categories. Not only do label transformers leave the rules well-formed (so that they never need reformulating when extending the described language), but also computations and bisimulation equivalence are preserved [43].

Table 4. Modular SOS

$e \in \mathit{Exp} ::= \ldots \mid V$
$s \in \mathit{Stm} ::= \ldots \mid ()$

$$\boxed{s \xrightarrow{\alpha} s'}$$

$$\frac{e \xrightarrow{\alpha} e'}{\texttt{if(}e\texttt{)}s \xrightarrow{\alpha} \texttt{if(}e'\texttt{)}s} \tag{6}$$

$$\texttt{if(}tt\texttt{)}s \xrightarrow{\iota} s \qquad \texttt{if(}f\!f\texttt{)}s \xrightarrow{\iota} (\,) \tag{7}$$

$$\boxed{e \xrightarrow{\alpha} e'}$$

$$\frac{e \xrightarrow{\alpha} e'}{x \texttt{=} e \xrightarrow{\alpha} x \texttt{=} e'} \tag{8}$$

$$x \texttt{=} v \xrightarrow{\alpha} v \quad \text{if } l = \iota.\rho(x), \quad \alpha = \iota[\sigma' = \iota.\sigma[v/l]] \tag{9}$$

▷ Similarly, Modular Natural Semantics requires all auxiliary entities to be incorporated in labels on evaluations.

Apart from the usual differences between SOS and natural semantics, observe in Table 5 that composition of labels has to be used explicitly in modular natural semantics. This composition corresponds to the threading of the store through premises of rules in conventional natural semantics, as was illustrated in Table 3. In fact sequential composition is not the only possibility for combining labels: when the labels of a modular natural semantics are taken to be finite sequences, it is possible to describe interleaving in terms of shuffling labels.

Table 5. Modular natural semantics

$e \in \mathit{Exp}$
$s \in \mathit{Stm}$

$$\boxed{s \xrightarrow{\alpha} (\,)}$$

$$\frac{e \xrightarrow{\alpha_1} tt \quad s \xrightarrow{\alpha_2} (\,)}{\texttt{if(}e\texttt{)}s \xrightarrow{\alpha} (\,)} \text{ if } \alpha = \alpha_1 \ ; \ \alpha_2 \qquad \frac{e \xrightarrow{\alpha} f\!f}{\texttt{if(}e\texttt{)}s \xrightarrow{\alpha} (\,)} \tag{10}$$

$$\boxed{e \xrightarrow{\alpha} v}$$

$$\frac{e \xrightarrow{\alpha_1} v}{x\texttt{=}e \xrightarrow{\alpha} v} \text{ if } l = \iota.\rho(x), \quad \alpha = \alpha_1 \ ; \ \iota[\sigma' = \iota.\sigma[v/l]] \tag{11}$$

Reduction Semantics was developed by Felleisen and his colleagues towards the end of the 1980's [16].

▷ Computations are modelled as sequences of term rewriting steps (reductions).

Here, both computed values and auxiliary entities are represented as terms: there is little or no separation between syntactic and semantic entities. The sequence of rewriting steps in a computation may be infinite.

▷ Reductions are restricted to occur in evaluation contexts, the general form of which is specified by a context-free grammar.

The unique hole in each context is indicated by '[ ]' in the grammar for the contexts *STM* and *EXP* in Table 6. Filling the hole of a context *A* by a tree *b* is written *A*[*b*]. Observe that the alternatives for contexts correspond to the inference rules for the same constructs in Table 2. Clearly, it is more concise to specify a grammar than a set of inference rules. Moreover, reduction semantics has the advantage over SOS that a reduction step may replace the *context* (as well as its contents), which can exploited not only to deal with effects on storage, but also with control constructs such as `call/cc`. Reduction semantics has the advantage over natural semantics that it can cope with non-terminating computations, as well as with synchronization and interleaving [52]; but it does not appear to have significant advantages over SOS, apart from greater conciseness.

Table 6. Reduction semantics

$$\begin{array}{lll} e \in & \mathit{Exp} & ::= \ldots \mid \texttt{true} \mid \texttt{false} \mid L \mid V \\ s \in & \mathit{Stm} & ::= \ldots \mid \{\} \\ S \in & \mathit{STM} & ::= [\,] \mid \texttt{if(}\mathit{EXP}\texttt{)}\,\mathit{Stm} \mid \ldots \\ E \in & \mathit{EXP} & ::= [\,] \mid L\,\texttt{=}\,\mathit{EXP} \mid \ldots \end{array}$$

$$\boxed{s \to s'}$$

$$\texttt{if(true)}\,s \to s \qquad \texttt{if(false)}\,s \to \{\} \tag{12}$$

$$\boxed{s, \sigma \longrightarrow s', \sigma'}$$

$$S[s], \sigma \longrightarrow S[s'], \sigma \quad \text{if } s \to s' \tag{13}$$

$$S[l\,\texttt{=}\,v], \sigma \longrightarrow S[v], \sigma[v/l] \tag{14}$$

Abstract State Machines (ASMs), previously called “evolving algebras” [20], is a framework proposed by Gurevich in the late 1980’s.

▷ Computations are modelled as sequences of parallel sets of assignments to values of particular functions on particular arguments.

The sequences may be finite or infinite. Dynamic function values remain stable when not updated; functions declared to be static never get updated after their initial definitions. The values of all the functions determine the state of the computation.

▷ States include control-flow graphs representing the entire program.

In the fragment shown in Table 7, the functions *fst*, *nxt* represent *normal* control flow between phrases. However, flow of control need not follow the structure of the program at all: in principle, the pointer *task*, normally indicating the next part of the program to be executed, can be set arbitrarily. The control-flow graph itself is static, but computed values can be associated with nodes by a separate dynamic function, such as *val* in Table 7. Scopes of bindings are represented indirectly, by explicit stacking of values, rather than by using environments or syntactic substitution.

Table 7. Abstract state machines

```
$task$     : $Phrase$
$fst, nxt$ : $Phrase \to Phrase$
$val$      : $Exp \to V$

let $s' = \texttt{if(}e\texttt{)}s$ in                                      (15)
    $fst(s') = fst(e), nxt(e) = s', nxt(s) = nxt(s')$

if $task$ is $\texttt{if(}e\texttt{)}s$ then                                (16)
  if $val(e)$ then $task := fst(s)$
              else $task := nxt(task)$

let $e' = (x \texttt{=} e)$ in                                              (17)
    $fst(e') = fst(e), nxt(e) = e'$

if $task$ is $(x \texttt{=} e)$ then                                        (18)
  $loc(x) := val(e)$,
  $val(task) := val(e)$,
  $task := nxt(task)$
```

Other Operational Frameworks

▷ Various other operational frameworks have been developed.

These include translation to code for the SECD abstract machine [31] and the VDL abstract machine [61], the SMoLCS framework [7], and the EOS framework [13].

2.2 Denotational Semantics

Apart from the original Scott-Strachey style of denotational semantics, we consider here also VDM, Monadic Semantics, and Predicate Transformers.

▷ Denotational Semantics models each part of a program as its *denotation*, representing its contribution to the overall behaviour of the enclosing program.

Denotations are typically higher-order functions between complete partial orders (cpos) [19]. The semantics of a complete program is its observable behaviour, which is obtained from its denotation.

▷ Semantic functions that map phrases to their denotations are defined inductively by sets of semantic equations, ensuring compositionality.

Such inductive definitions correspond to so-called initial algebra semantics [18]. The semantics of loops and recursion usually involves the explicit use of fixed-point operators, although some authors prefer to specify these as equations whose (least) solution is to be found.

Scott-Strachey Semantics is the original style of denotational semantics, developed by Scott and Strachey at the end of the 1960's [40,48,54,56,57].

▷ Domains of denotations and auxiliary entities are defined by domain equations.

The domains are usually ω-complete partial orders (cpos); only *continuous* functions between domains are considered. Domain equations always have "least" solutions (up to isomorphism), e.g. $D = N + [D \to D]$ defines a domain D including both the natural numbers and all continuous functions on D. The elements of domains are specified in typed λ-notation.

▷ Typically, denotations are functions of environments, continuations, and stores.

Many standard techniques for representing programming concepts as pure mathematical functions have been established. For instance, sequencing may be represented either by composition of strict functions, or by use of *continuations*; the latter are illustrated in Table 8. The denotational description of nondeterminism, concurrency, and interleaving requires the use of power domains (and a significant amount of extra notation).

Table 8. Scott-Strachey semantics

$$
\begin{aligned}
\rho \in \mathit{Env} &= \mathit{Var} \to V \\
\sigma \in S &= L \to V \\
\theta \in C &= S \to A \\
\kappa \in K &= V \to C \\
\mathcal{S} :\ & \mathit{Stm} \to \mathit{Env} \to C \to C \\
\mathcal{E} :\ & \mathit{Exp} \to \mathit{Env} \to K \to C
\end{aligned}
$$

$$\mathcal{S}[\![\texttt{if(}e\texttt{)}s]\!] = \lambda\rho.\lambda\theta.\mathcal{E}[\![e]\!]\rho(\lambda v.v|B \to \mathcal{S}[\![s]\!]\rho\theta, \theta) \tag{19}$$

$$\mathcal{E}[\![x\,\texttt{=}\,e]\!] = \lambda\rho.\lambda\kappa.\mathcal{E}[\![e]\!]\rho(\lambda v.\lambda\sigma.\kappa(v)(\sigma[v/\rho(x)|L])) \tag{20}$$

VDM Semantics is a notational variant of denotational semantics, developed by Bjørner and C. B. Jones in the mid-1970's [9] for use in software development.

▷ The meta-notation Meta-IV used in VDM provides extra generality regarding abstract syntax.

Lists, sets, and maps may be used as components of abstract syntax trees. This allows some semantic properties, such as the insignificance of the order of declarations in certain languages, to be made evident in the abstract syntax.

▷ It also ensures propagation of effects and exceptions.

Meta-IV provides notation for sequencing effects on stores, the use of which is illustrated in Table 9, and for exception-handling.

Table 9. VDM semantics

$$\mathcal{M} : \mathit{Stm} \to \mathit{ENV} \to S \overset{\sim}{\to} S$$
$$\mathcal{M} : \mathit{Exp} \to \mathit{ENV} \to S \overset{\sim}{\to} S \times V$$

$$\begin{aligned}\mathcal{M}[\mathit{mk\text{-}If}(e,s)](\rho) = {} & \text{def } v : \mathcal{M}[e](\rho); \\ & \text{if } v \text{ then } \mathcal{M}[s](\rho) \text{ else } I_S\end{aligned} \tag{21}$$

$$\begin{aligned}\mathcal{M}[\mathit{mk\text{-}Assign}(x,e)](\rho) = {} & \text{def } l : \rho(x); \\ & \text{def } v : \mathcal{M}[e_2](\rho); \\ & \text{assign}(l,v);\ \text{return}(v)\end{aligned} \tag{22}$$

Monadic Semantics was developed by Moggi at the end of the 1980's [37,38], based on category-theoretic concepts.

▷ Denotations in Monadic Semantics are elements of *monads*, and their composition is expressed independently of the domains used to construct the monads.

Intuitively, a monad corresponds to a particular notion of computation. Various notations for monadic composition are available. In the one whose use is illustrated in Table 10, 'let $v = c_1$ in c_2' expresses that the computation c_1 is performed first; the computed value is then referred to as v in the computation c_2. In so-called exception monads, raising an exception in c_1 may cause c_2 to be skipped; various other monads define composition in different ways. The use of the composition notation depends only on knowing the type of value computed by c_1, not on the structure of the domain of computations in any particular monad. The notation '$[v]$' expresses the trivial computation that merely returns the value v. Comparing Tables 9 and 10, the closeness of the monadic notation to that provided a decade earlier by Meta-IV is quite striking.

▷ Monad transformers construct monads *incrementally*, and *lift* the associated functions.

A monad transformer adds a new aspect of computation to a given monad. The order of applying the transformers can be critical—in particular, the transformers that provides continuations does not commute with other transformers. Moreover, not all functions can be lifted uniformly through monad transformers.

Moggi has subsequently developed a more general (and rather less category-theoretic) framework based on translation between meta-languages [39].

Table 10. Monadic semantics

$$\mathcal{S} : \mathit{Stm} \to T()$$
$$\mathcal{E} : \mathit{Exp} \to T(V)$$
$$\mathcal{S}[\![\texttt{if(}e\texttt{)}s]\!] = \text{let } v = \mathcal{E}[\![e]\!] \text{ in case } v|B \text{ of } (\mathit{tt} \Rightarrow \mathcal{S}[\![s]\!] \mid \mathit{ff} \Rightarrow [()]) \tag{23}$$
$$\mathcal{E}[\![x\,\texttt{=}\,e]\!] = \text{let } l = \mathit{lookup}(x) \text{ in let } v = \mathcal{E}[\![e]\!] \text{ in let } () = \mathit{assign}(l, v) \text{ in } [v] \tag{24}$$

Predicate Transformer Semantics was proposed by Dijkstra in the mid-1970's [15]. It is often regarded as axiomatic, since it involves assertions about the values of variables before and after executing statements. However, it is more properly classified as denotational.

▷ The denotation of a phrase is a predicate transformer that returns the weakest condition which ensures termination of the phrase with the argument condition holding.

Predicate transformers are required to have properties corresponding to continuity of functions on Scott-domains. The description of assignment involves substitution in formulae. Expression evaluation is assumed to be free of side-effects, errors, and non-termination, so that boolean expressions may be used as formulae, and expressions substituted syntactically for variables. In the illustrations given below and in Sect. 2.3, we treat only assignment *statements* of the form $x\,\texttt{=}\,e\texttt{;}$, and assume that assignment *expressions* cannot occur elsewhere.

Table 11. Predicate transformer semantics

$$\mathit{wp} : \mathit{Stm} \times \mathit{Pred} \to \mathit{Pred}$$
$$\mathit{wp}(\texttt{if(}e\texttt{)}s, Q) \Leftrightarrow (e \Rightarrow \mathit{wp}(s, Q)) \wedge (\neg\, e \Rightarrow Q) \tag{25}$$
$$\mathit{wp}(x\,\texttt{=}\,e\texttt{;}, Q) \Leftrightarrow Q[e/x] \tag{26}$$

Other Denotational Frameworks include Naive Denotational Semantics [10], partially-additive semantics [32], use of metric spaces [4], and Extensible Denotational Semantics [11]. Further variants of denotational semantics involve natural transformations between categories of stores [49], Linear Logic and games [1].

2.3 Axiomatic Semantics

Here we consider both Hoare Logic and Algebraic Semantics.

▷ Axiomatic semantics restricts the potential models by asserting properties.

As usual with axiomatic specifications, it may be unclear whether sufficiently many properties have been given (if not, there may be more than one model), or whether an inconsistent set of properties has been given (then there are no models at all, cf. [5]). As with predicate transformers, boolean expressions are used as formulae, and the semantics of assignment statements involves substitution of expressions for variables, so expressions cannot have side-effects.

Hoare Logic was developed in the late 1960's [26].

▷ Partial correctness assertions $P\{s\}R$ are specified by axioms and inference rules.

The partial correctness assertion $P\{s\}R$ requires that if P holds and the subsequent execution of the statement s terminates, then R holds. A general rule, applicable to all statements, is included in Table 12 to allow strengthening of pre-conditions and weakening of post-conditions.

Table 12. Hoare logic

$$\frac{(P \wedge e)\{s\}R \qquad (P \wedge \neg\, e \Rightarrow R)}{P\{\texttt{if(}e\texttt{)}s\}R} \qquad P[e/x]\{x\,\texttt{=}\,e\,\texttt{;}\}P \tag{27}$$

$$\frac{P' \Rightarrow P \qquad P\{s\}R \qquad R \Rightarrow R'}{P'\{s\}R'} \tag{28}$$

Algebraic Semantics for programming languages was developed by Hoare and his colleagues at Oxford in the 1990's [27].

▷ Equations and inclusions between phrase terms characterize the relationship between their interpretations.

It appears that this approach is applicable only to languages that have a particularly expressive syntax and clean semantics.

Other Axiomatic Frameworks include Dynamic Logic [23].

Table 13. Algebraic semantics

$$\texttt{if(}e\texttt{)}\,P = (e \to P \;[]\; \neg\, e \to \mathit{skip}) \tag{29}$$
$$e \to P = e^{\top};\; P \tag{30}$$

2.4 Complementary Semantics

▷ Particular semantic frameworks may have advantages for different uses.

It has been proposed [27,28,54] that one should give not just one but several complete descriptions of the same language, using different kinds of frameworks.

▷ The different descriptions of a language should be consistent.

It is well known that it is difficult to prove consistency between semantic descriptions in different frameworks. To derive one description from another, for example by use of abstract interpretation [12], may be more feasible, and help to ensure consistency.

2.5 Hybrid Semantics

▷ A hybrid approach to semantics involves more than one framework in the same description.

An analogous example of a hybrid approach in descriptions of syntax is the use of regular grammars to describe lexical symbols, and of context-free grammars to describe phrase structure with lexical symbols as terminal symbols.

▷ The various semantic frameworks may have advantages for different levels of complete semantic descriptions.

The separation of semantics into static and dynamic phases encourages a hybrid approach to complete semantic descriptions; below, we consider approaches that involve hybrid descriptions also within dynamic semantics.

Action Semantics was developed by the present author, in collaboration with Watt, in the second half of the 1980's [41,47,59].

▷ Action Semantics is a hybrid of denotational and operational semantics.

As in denotational semantics, inductively-defined semantic functions map phrases to their denotations, only here, the denotations are so-called *actions*; the notation for actions is itself defined operationally [41,45].

▷ Action semantics avoids the use of higher-order functions expressed in lambda-notation.

The universe of pure mathematical functions is so distant from that of (most) programming languages that the representation of programming concepts in it is often excessively complex. The foundations of reflexive Scott-domains and higher-order functions are unfamiliar and inaccessible to many programmers (although the idea of functions that take other functions as arguments, and perhaps also return functions as results, is not difficult in itself). The use of pure lambda-notation has some pragmatic drawbacks, especially concerning modularity; the monadic style of denotational semantics was developed (partly) to circumvent these.

▷ Action semantics provides a rich action notation with a direct operational interpretation.

The universe of actions involves not only control and data flow, but also scopes of bindings, effects on storage, and interactive processes, allowing a simple and direct representation of many programming concepts. The foundations of action notation involve SOS and algebraic specifications, which are both generally regarded as more accessible than domain theory.

Table 14. Action semantics

$$\begin{aligned}
&\mathit{execute} : \mathit{Stm} \to \mathit{action}[\mathit{giving}()]\\
&\mathit{evaluate} : \mathit{Exp} \to \mathit{action}[\mathit{giving\ a\ value}]
\end{aligned}$$

$$\begin{aligned}\mathit{execute}[\![\texttt{if(}e\texttt{)}s]\!] = {}& \mathit{evaluate}[\![e]\!]\ \mathit{then}\\ &(\ \mathit{given\ true\ then\ execute}[\![s]\!]\ \mathit{otherwise\ skip}\ )\end{aligned} \tag{31}$$

$$\begin{aligned}\mathit{evaluate}[\![x\,\texttt{=}\,e]\!] = {}& \mathit{give\ the\ variable\ bound\ to}\ x\ \mathit{and\ evaluate}[\![e]\!]\\ &\mathit{then}\ (\ \mathit{assign\ and\ give\ the\ value\#2}\ )\end{aligned} \tag{32}$$

Other Hybrid Frameworks include Modular Denotational Semantics [17], Modular Monadic Action Semantics [58], Type-Theoretic Interpretation [24], and translation between meta-languages [39]. As mentioned earlier, various operational frameworks such as VDL may be considered as hybrids.

3 Potential and Actual Uses

We have seen that many different semantic frameworks have been developed. Let us now consider the main *potential* uses of formal semantic descriptions, and list some of the *actual* uses.[3]

[3] The author would be grateful for reports of further cases of practical use.

3.1 Potential Uses of Semantic Descriptions

▷ Formal semantic descriptions may be used by language designers to record their decisions.

Formal grammars are useful for recording proposals and decisions concerning (concrete and abstract) syntax during the language design process; semantic descriptions could be just as useful for recording the intended semantics of the constructs concerned. In cases where the language designers do not formulate a proper semantic description, their familiarity with semantic concepts may still be beneficial to the design process, of course.

▷ Such descriptions may also be used in the documentation of a completed language design.

A reference manual or standards document for a programming language should provide complete and unambiguous descriptions of both the syntax and the semantics of the language. Unofficial semantic descriptions contributed by "outsiders", however, may often be of little real use (except for demonstrating that particular frameworks can be used to describe particular languages) and will be ignored here.[4]

▷ Formal semantic descriptions may be used to derive implementations.

Ideally, it would be possible to generate an efficient compiler or interpreter automatically from a semantic description,—just as scanners and parsers can be generated from grammars. Even generation of inefficient implementations may be adequate for rapid prototyping purposes, and for languages where efficiency is of no concern. Formal semantic descriptions may also be the basis for systematic (manual) development of implementations.

▷ They may be used in connection with program verification or development.

One of the primary advantages of formal semantic descriptions is that they provide a basis for sound reasoning about program behaviour and equivalence.

3.2 Actual Uses of Semantic Descriptions

Operational Semantics:

SOS was used during the design of Facile (a higher-order, concurrent language). The SMoLCS variant of SOS was used to describe full Ada in an official project. (SOS has also been much used to describe process calculi, including LOTOS, but here, let us restrict our attention to descriptions of languages that are normally used for programming rather than specification.)

[4] The questions that they often raise about details of language design may be prompted by any close and systematic study.

Natural Semantics was used during the design of Standard ML (SML), and to give an official definition of the language. The ML Tool-Kit implementation of SML was derived systematically from the official definition.

Reduction Semantics was used during the design of Concurrent ML (CML). However, the formal semantic description of CML has apparently not been incorporated in the CML reference manual—nor is there even a link to it from the CML home page.

ASM was adopted by ISO for use in the Prolog standard, and by ITU for use in the standard for SDL-2000 (the description of the latter being however in quite a different style from that previously used in ASM specifications). Also, a recent book including ASM specifications of Java and the JVM deserves mention as "the most comprehensive and consistent formal account of Java and the JVM, to date" (although it does not appear to have received any endorsement from the language developers). An ASM specification of a language for *Universal Plug and Play* (UPnP) is apparently being used by developers at Microsoft. ASM descriptions of Prolog and occam have been used as a basis for program verification and the derivation of implementations.

Denotational Semantics:

Scott-Strachey semantics was used during the design of Ada, and the resulting description (of sequential Ada only) was included as part of the official language documents. The same style of semantics was used by the designers of the functional programming language Scheme, and subsequently incorporated in the Scheme reference manual.

VDM was used in the CCITT standard for the telecommunication language CHILL, and in the ISO standards for SDL-88, SDL-92, and SDL-96. A successful Ada compiler was derived systematically from a VDM description of the full language.

Monadic semantics appears to be currently lacking an example of its use in connection with a practical programming language.

Axiomatic Semantics:

Hoare Logic was used during the design of Pascal, and the resulting description as the basis for program verification,

Algebraic Semantics influenced the design of occam, and the resulting algebraic semantics of occam itself may be regarded as part of the reference material for the language. Algebraic semantics has also been used as the basis for the development of a correct compiler for occam.

Hybrid Semantics:

Action Semantics was adopted to describe the language ANDF [22] (a language intended for use in software package distribution).

Compared to the amount of effort that has been devoted to the development of various semantic frameworks over more than three decades, and all the potential uses listed above, the list of actual uses may be considered as relatively disappointing. Why has there not been far greater use of formal semantics in practice? Some—or perhaps even all—of the factors listed in the following section may be relevant.

4 Hindrances to Greater Use

▷ Many frameworks for semantics are not particularly user-friendly.

In some frameworks, the familiar programming concepts underlying a described language (such as flow of control and scopes of bindings) are not indicated by fixed symbols, but rather get encoded in patterns of use of general notation. For instance, order of evaluation may be specified by patterns of transitions in inference rules in SOS or Natural Semantics, and by use of patterns of lambda-notation in Denotational Semantics. This reduces comprehensibility (at least until the relevant patterns have been learned). A related potential hindrance is when the details of mathematical foundations are reflected directly in semantic descriptions, since those foundations may seem inaccessible to many potential users [55].

Practical use of formal semantics involves descriptions of major programming languages. Few frameworks scale up smoothly from the tidy illustrative languages described in text-books and papers to languages such as C and Java—even Standard ML, a clean language designed by theoreticians, turned out to be a considerable challenge to describe accurately in Natural Semantics. Tool support could alleviate the problems of writing, checking, and navigating in large-scale descriptions, but mature tools are totally lacking for most frameworks, in marked contrast to the sophisticated programming environments available to programmers [25].

▷ Much effort is required to develop a complete semantic description, but the potential benefits are somewhat intangible.

For theoreticians, there is generally little academic reward for producing a semantic description of a full programming language (unless it is so interesting that it can be published as a book). Merely overcoming pragmatic problems in applications of semantics is not of much theoretical interest. Practitioners involved with designing and implementing a programming language have to weigh the effort of formulating a formal semantics against how much practical advantage it gives. In any case, the number of potential users for any particular semantic description is generally very small.

▷ In contrast, formal *syntax* has become quite popular.

One reason for the popularity of formal grammars may be that BNF—at least when extended with notation for iteration and optional phrases—is exceptionally user-friendly. The major relevant concepts (alternatives, sequencing, and iteration) are all expressed directly, and the notational variation concerning iteration, although irritating, does not impede reading and understanding grammars. BNF scales up smoothly to descriptions of larger languages. Moreover, *practical* use of formal syntax is strongly supported by tools, both for prototyping grammars (e.g., to check that a grammar is actually LALR(1)) and for generating useful parsers.

▷ Formal semantic descriptions should aim for the *practicality* of BNF...

Denotational semantics espoused the idea of extending BNF to semantics, but the good pragmatic aspects of BNF have been sadly lacking in denotational descriptions (at least during the 1970's and 1980's, before the development of the monadic approach); most of the other frameworks surveyed in Sect. 2 suffer from similar problems with pragmatic aspects, especially regarding lack of tool support.

5 Conclusion

▷ A large number of semantic frameworks have been provided during the past three decades.

We have classified them mainly as operational, denotational, and axiomatic. To give complementary semantic descriptions of the same language in different frameworks (and prove them consistent) appears to be much too demanding, at least for the average semanticist. It seems preferable to exploit the best features of the various frameworks synergistically, in different parts of a single hybrid description.

▷ There are plenty of potential uses for formal semantic descriptions, but disappointingly few actual practical applications of real significance (so far).

Language designers seldom use formal semantic descriptions at all during the design process [60]. In only a few cases has a formal semantics of a language has been provided for reference purposes, and implementors have referred to it for clarification.

▷ The main hindrances to greater use of formal semantics appear to be lack of user-friendliness, and lack of tool support.

Semantic descriptions in many frameworks have poor pragmatic aspects, for instance concerning modularity and the smoothness of scaling up to descriptions of larger languages. Quality tools are badly needed to assist the writing, checking, and reading of semantic descriptions. *Meta-systems* such as the ASF+SDF Meta-Environment and Maude may support the development of such tools for some semantic frameworks.

▷ Development of a user-friendly semantic framework allowing generation of efficient compilers should encourage practical use.

Semantics-directed compiler generation was a popular topic for PhD theses in the 1980's and the first half of the 1990's. Despite the construction of some promising prototype systems, no tools for producing efficient compilers directly from semantic descriptions appear to be currently available.[5] To have such tools would not only provide a real incentive for writing semantic descriptions, they would also allow the empirical testing of whether the semantics really does define the intended language.

▷ The ASM approach is a good candidate for such a framework.

As may be seen from the ASM web pages at `http://www.eecs.umich.edu/gasm`, the ASM approach has a large and energetic following, and has already made a considerable impact regarding practical applications—although compiler generation does not yet appear to be one of its strengths.

▷ Action Semantics also looks promising.

Action semantics was designed with user-friendliness as first priority; it was chosen for use in the description of ANDF because of its good pragmatic qualities [21]. Substantial experience with prototype systems for compiler and interpreter generation based on action semantics has already been obtained. A new, significantly simplified version of action semantics is currently about to be released—simple enough to be taught at undergraduate level. Not many people are working on action semantics at present. Readers who might be interested in helping to produce useful tools for action semantics, especially compiler generators, are cordially invited to take a closer look at the framework from the materials available via the home page at `http://www.brics.dk/Projects/AS`, and to contact the author if wanting to help.

Acknowledgements. Thanks to Yuri Gurevich, Jan Heering, Cliff Jones, Paul Klint, John Reynolds, Dana Scott, and several participants at PSI'01, for critical comments on the preliminary version of this paper, "What Use Is Formal Semantics?"; and to Bartek Klin for suggesting improvements to a draft of the revised version. The responsibility for remaining infelicities in the contents or presentation is entirely the author's. Thanks also to the organizers of PSI'01 for the invitation to speak on this topic, and to the editors of these Proceedings for their patience. The author gratefully acknowledges the support of BRICS (Basic Research in Computer Science, funded by the Danish National Research Foundation).

[5] The RML system [50] is however able to produce efficient *interpreters* from (a special form of) natural semantics.

References

1. S. Abramsky and G. McCusker. Linearity, sharing, and state: A fully abstract game semantics for Idealized Algol with active expressions (extended abstract). In *Proc. 1996 Workshop on Linear Logic*, volume 3 of *Electronic Notes in TCS*. Elsevier, 1996.
2. L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 3, pages 197–291. Elsevier Science, 2001.
3. C. Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.
4. P. America, J. de Bakker, J. N. Kok, and J. J. M. M. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–206, 1989.
5. E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on 'Program proving: Jumps and functions, by M. Clint and C. A. R. Hoare'. *Acta Inf.*, 6:317–318, 1976.
6. E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.
7. E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *TAPSOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa*, volume 249 of *LNCS*. Springer-Verlag, 1987.
8. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. Frontier Series. ACM Press, 1989.
9. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.
10. A. Blikle and A. Tarlecki. Naive denotational semantics. In *Information Processing 83, Proc. IFIP Congress 83*. North-Holland, 1983.
11. R. Cartwright and M. Felleisen. Extensible denotational semantics specifications. In *TACS'94, Proc. Symp. on Theoretical Aspects of Computer Software, Sendai, Japan*, volume 789 of *LNCS*, pages 244–272. Springer-Verlag, 1994.
12. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *POPL'92, Proc. 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Albuquerque, New Mexico*, pages 84–94, 1992.
13. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.
14. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
15. E. W. Dijkstra. Guarded commands, non-determinacy, and formal derivations of programs. *Commun. ACM*, 18:453–457, 1975.
16. M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.
17. J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Proc. Int. Coll. on Formalization of Programming Concepts, Peñiscola*, number 107 in LNCS. Springer-Verlag, 1981.
18. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.
19. C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 12. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

20. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
21. B. S. Hansen and J. Bundgaard. The role of the ANDF formal specification. Technical Report 202104/RPT/5, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1992.
22. B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In [42], pages 34–42, 1994.
23. D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II. D. Reidel Publishing Company, 1984.
24. R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998.
25. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, Mar. 2000.
26. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.
27. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
28. C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Act Inf.*, 3:135–153, 1974.
29. G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
30. U. Kastens, P. Pfahler, and M. Jung. The eli system. In *CC'98, Proceedings 7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 294–297. Springer-Verlag, 1998.
31. P. J. Landin. A formal description of Algol60. In *Formal Language Description Languages for Computer Programming, Proc. IFIP TC2 Working Conference, 1964*, pages 266–294. IFIP, North-Holland, 1966.
32. E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Springer-Verlag, 1986.
33. J. McCarthy. Towards a mathematical science of computation. In *Information Processing 62, Proc. IFIP Congress 62*, pages 21–28. North-Holland, 1962.
34. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
35. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
36. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1997.
37. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
38. E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
39. E. Moggi. Metalanguages and applications. In *Semantics and Logics of Computation*, Publications of the Newton Institute. CUP, 1997.
40. P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
41. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

42. P. D. Mosses, editor. *AS'94, Proc. 1st Intl. Workshop on Action Semantics, Edinburgh*, number NS-94-1 in Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. `http://www.brics.dk/NS/94/1/BRICS-NS-94-1/`.
43. P. D. Mosses. Foundations of modular SOS. Research Series RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
`http://www.brics.dk/RS/99/54`; full version of [44].
44. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at `http://www.brics.dk/RS/99/54/`.
45. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at
`http://www.brics.dk/RS/99/56/`.
46. P. D. Mosses. A modular SOS for ML concurrency primitives. Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
`http://www.brics.dk/RS/99/57/`.
47. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.
48. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, Chichester, UK, 1992.
49. F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.
50. M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *LNCS*. Springer-Verlag, 1999.
51. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
52. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.
53. J. R. X. Ross. *An Evaluation Based Approach to Process Calculi*. PhD thesis, University of Cambridge, 1999.
54. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.
55. D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1), 1997.
56. D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.
57. R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19:437–453, 1976.
58. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.
59. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.
60. D. A. Watt. Why don't programming language designers use formal methods? In R. Barros, editor, *Anais XXIII Seminário Integrado de Software e Hardware*, pages 1–16, UFPE, Recife, Brazil, 1996.
61. P. Wegner. The Vienna definition language. *ACM Comput. Surv.*, 4:5–63, 1972.

Binding-Time Analysis for Polymorphic Types

Rogardt Heldal and John Hughes

Chalmers University of Technology, S-41296 Göteborg
heldal@cs.chalmers.se, www.cs.chalmers/~heldal.

Abstract. Offline partial evaluators specialise programs with annotations which distinguish *specialisation-time* (or static) computations from run-time ones. These annotations are generated by a *binding-time analyser*, via type inference in a suitable type-system. Henglein and Mossin developed a type system which allows *polymorphism* in binding-times, so that the same code can be specialised with different computations being static at different uses. We extend their work to permit polymorphism in *types* as well. This is particularly important for separately compiled libraries.
Following Henglein and Mossin, binding-times are passed as parameters during specialisation, but types are not. Instead, we pass *coercion functions* when necessary, which makes specialisation less "interpretive" than it would otherwise be. We keep track of the coercions needed by assigning *qualified types* to polymorphic functions.
We also consider hand-annotations to provide limited user control over the binding-time analysis, which our prototype implementation showed to be necessary.

1 Introduction

Partial evaluation is by now a well-established technique for specialising programs [14], and practical tools have been implemented for a variety of programming languages [2,1,18,5]. Our interest is in partial evaluation of modern typed functional languages, such as ML [20] or Haskell [16]. One of the key features of these languages is polymorphic typing [19], yet to date the impact of polymorphism on partial evaluation has not been studied. In this paper we explain how to extend an offline partial evaluator to handle a polymorphic language.

1.1 Background: Polymorphism

A *polymorphic function* is a function which may be applied to many different types of argument. In ML and Haskell, the types of such functions are expressed using a "forall" quantifier: for example, the well-known *map* function, which applies a function to every element of a list, has the type

$$map :: \forall \alpha_1, \alpha_2.(\alpha_1 \to \alpha_2) \to [\alpha_1] \to [\alpha_2]$$

meaning that for *any* types α_1 and α_2, *map* takes a function of type $\alpha_1 \to \alpha_2$ and a list of type $[\alpha_1]$, and produces a list of type $[\alpha_2]$. ($[\alpha]$ is our notation for a list type with elements of type α).

Polymorphic functions are heavily used in real functional programs. In particular, library functions are frequently polymorphic, since the types at which they will be needed are not known when the library is written. The standard library contains many polymorphic functions such as *map* and *foldr* (which takes a binary operator and its unit, and combines the elements of a list using the operator). These polymorphic functions greatly simplify programming, for example, the sum of a list of integers *xs* can be computed as *foldr* $(+)$ 0 *xs*, and the conjunction of a list of booleans *bs* can be computed as *foldr* $(\wedge)$ **true** *bs*.

1.2 Background: Partial Evaluation

A partial evaluator is a tool which takes a program and a *partially known* input, and performs operations in the program which depend only on the known parts, generating a specialised program which processes the remainder. For example, specialising *foldr* to the inputs *foldr* $(+)$ 0 $[x, y, z]$, where x, y and z are unknown, would generate the specialised program $x + y + z + 0$. Here the construction of the known list, and the recursion over it inside *foldr*, have been performed by the partial evaluator: only the actual computations of the sum of the unknown quantities remains in the specialised code.

Partial evaluators can be classified into *online* and *offline*. Online partial evaluators decide dynamically during specialisation which operations to perform, and which to build into the residual program: an operator is performed if its operands are known in that particular instance. An offline partial evaluator processes an *annotated* program, in which the annotations determine whether an operator is to be applied or not. Offline partial evaluators are generally more conservative, but simpler and more predictable; we focus on this type in this article.

As an example, we annotate the *power* function, which computes x^n, for specialisation with a known value for n. We annotate each operator with a *binding-time*, S (static) or D (dynamic), and we write function application explicitly as @ so that we can annotate it. Operators annotated static are performed during partial evaluation.

$$\begin{aligned} \mathit{power}\ n\ x = {} & \mathbf{if}^S\ n =^S 0 \\ & \mathbf{then}\ \mathit{Int}^{SD}\ 1 \\ & \mathbf{else}\ x \times^D \mathit{power}@^S(n -^S 1)@^S x \end{aligned}$$

In annotated programs we distinguish between known static values, and the corresponding dynamic code fragment; in this example, since the result of specialising *power* is code, the *coercion* Int^{SD} must be used to convert the static integer 1 to the correct type.

Annotated programs can be *interpreted* by a partial evaluator, or *compiled* into a *generating extension*. This is a program which, given the partially known input, generates a specialised version of the annotated program directly. The generating extension of this annotated *power* function is itself a recursive function, which computes the static operations directly, and generates code for the dynamic ones. Running the generating extension with the arguments 3 and "x" (a code fragment) produces the code fragment "$x \times x \times x \times 1$". Notice that, in the generating extension, a static integer and a dynamic integer are represented by different types: the former by an integer, and the latter by a code fragment — for example, an abstract syntax tree. Thus coercions do real work.

However, fixed annotations work poorly in large programs. Library functions in particular may be called in many contexts, with combinations of static and dynamic arguments which are unknown at the time the function definition is annotated. This motivates *polychronic* annotations[1] containing binding-time *variables*, which are passed as parameters to annotated functions [7]. Using polychronic annotations, we can annotate the *power* function as

$$\begin{aligned}\mathit{power}\ \beta_1\ \beta_2\ n\ x = {} & \mathbf{if}^{\beta_1}\ n =^{\beta_1} \mathit{Int}^{S\beta_1}\ 0\\ & \mathbf{then}\ \mathit{Int}^{S(\beta_1\sqcup\beta_2)}\ 1\\ & \mathbf{else}\ x \times^{\beta_1\sqcup\beta_2} \mathit{power}\ \beta_1\ \beta_2 @^S (n -^{\beta_1} \mathit{Int}^{S\beta_1}\ 1) @^S x\end{aligned}$$

where the least upper bound of two binding times is determined by $S \leq D$. This version can be specialised to any combination of known and unknown arguments, but binding-times must actually be computed and passed as parameters in the generating extension, increasing the cost of specialisation somewhat. (We need not annotate the applications to binding-times, since these are always performed during specialisation). Notice also that many more coercions are needed, now that the binding-times are no longer known a priori.

The binding-time behaviour of this function can be captured by a *binding-time type*,

$$\mathit{power} :: \forall\beta_1,\beta_2.\mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_1\sqcup\beta_2}$$

in which each type constructor is annotated to indicate whether the corresponding value is known. Program annotations can be generated by inferring these types using a binding-time type system. Types must always be *well-formed*, in the sense that no static type appears under a dynamic type constructor.

2 What about Polymorphism?

When we try to incorporate polymorphic functions into this framework, we immediately run into difficulties. Consider, for example, a possible binding-time type for the *map* function:

$$\mathit{map} :: \forall\alpha_1,\alpha_2.\forall\beta_1,\beta_2.(\alpha_1 \to^{\beta_1} \alpha_2) \to^S [\alpha_1]^{\beta_2} \to^S [\alpha_2]^{\beta_2}$$

But not every instantiation of this type is well-formed: if either β_1 or β_2 is D, then neither α_1 nor α_2 may be instantiated to a static type, since this would produce an ill-formed type containing a static type under a dynamic type constructor. To capture such dependencies between variables, we add *constraints* to our binding-time types, which all instantiations must satisfy. Writing $\beta \rhd \alpha$ for the constraint

[1] Also, confusingly, called "polymorphic".

that if β is D, then α must be a dynamic type, we can give a correct type for *map* as

$$\begin{aligned}\mathit{map} :: \forall\alpha_1,\alpha_2.\forall\beta_1,\beta_2.(\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_1, \beta_2 \rhd \alpha_2) \Rightarrow \\ (\alpha_1 \to^{\beta_1} \alpha_2) \to^{S} [\alpha_1]^{\beta_2} \to^{S} [\alpha_2]^{\beta_2}\end{aligned}$$

These constraints have been used before [7], but did not appear in binding-time types since that paper did not consider polymorphism.

Now consider an even simpler polymorphic function,

$$\mathit{twice}\ f\ x = f@(f@x)$$

The standard type of this function is $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$, but for the purposes of specialisation we can be more liberal: we can allow the argument and result of f to have different binding-time types, provided the result can be coerced to the argument type. Thus we also need a *coercion* or *subtyping* constraint $\alpha_1 \leq \alpha_2$, which lets us give *twice* the binding-time type

$$\mathit{twice} :: \forall\alpha_1,\alpha_2.\forall\beta.(\beta \rhd \alpha_1, \beta \rhd \alpha_2, \alpha_2 \leq \alpha_1) \Rightarrow (\alpha_1 \to^{\beta} \alpha_2) \to^{S} \alpha_1 \to^{S} \alpha_2$$

However, there is more than one way that we might choose to annotate the *definition* of *twice*.

We might expect that, just as we pass binding-times explicitly in annotated programs, we should pass types explicitly to annotated polymorphic functions. Annotating *twice* in this way would result in something like

$$\mathit{twice}\ \alpha_1\ \alpha_2\ \beta\ f\ x = f@^{\beta}([\alpha_2 \mapsto \alpha_1]\ (f@^{\beta}x))$$

But notice that we need a coercion, which we have written as $[\alpha_2 \mapsto \alpha_1]$, between two unknown types here! The compiled code for a generating extension will need to construct representations of types during specialisation, pass them as parameters, and interpret them in order to implement such coercions. Because types may be complex, this may be expensive, and in any case we prefer to avoid interpretation in generating extensions.

Therefore, we treat polymorphic functions differently. Rather than passing *types* as parameters, we pass the necessary *coercion functions*, one for each subtype constraint in the function's type. With this idea, the annotated version of *twice* becomes

$$\mathit{twice}\ \beta\ \xi\ f\ x = f@^{\beta}(\xi\ (f@^{\beta}x))$$

where ξ implements the coercion $\alpha_2 \leq \alpha_1$. At each call of *twice*, we can pass a specialised coercion function for the types which actually occur.

3 Binding-Time Analysis

Binding-time annotations are usually constructed automatically by a *binding-time analyser.* We specify our polymorphic binding-time analysis via a type system for annotated programs, which guarantees that operations annotated as static never depend on dynamic values. Given an unannotated program, the binding-time analyser finds well-typed annotations that make as many operations as possible static. This type-based approach builds on earlier work by Dussart, Henglein and Mossin [13,7], which has been adopted for the Similix partial evaluator [2]. We favour a type-based approach because it is efficient, comprehensible, and extends naturally to handle polymorphism.

We shall specify the binding-time type system for the smallest interesting language, and then discuss how it is used to infer annotations.

3.1 The Binding-Time Type System

We consider an annotated λ-calculus with polymorphic **let** and one base type:

$$\begin{array}{ll} e[\mathit{Expression}] & ::= c \mid x \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \lambda x.e \mid e@^b\phi\ e \mid \lambda\beta.e \mid e\ b \mid \lambda\xi.e \mid e\ \phi \\ b[\mathit{Binding\text{-}time}] & ::= S \mid D \mid \beta \mid b \sqcup b \\ \phi[\mathit{Coercion}] & ::= \iota \mid \xi \mid \mathit{Int}^{bb} \mid \phi \rightarrow^{bb} \phi \end{array}$$

Here β is a binding-time variable, ξ is a coercion variable, x is a program variable, and c is a constant.

In this simple language, only function application need be annotated with a binding-time, and only function arguments need be coerced. Constants and λ-expressions are always static, and are coerced to be dynamic where necessary. **let**-expressions are always dynamic, but their bodies may even so be static since we use Bondorf's CPS specialisation [3], which moves the context of a **let** into its body, where it can be specialised. Applications to binding-times and coercions always take place during specialisation, and so need no annotation.

We have already seen integer coercions. A coercion $\phi_1 \rightarrow^{b_1 b_2} \phi_2$ coerces a function with binding-time b_1 to one with binding-time b_2, applying coercion ϕ_1 to the argument and ϕ_2 to the result. ι is the identity coercion.

Binding-time types and constraints take the form

$$\begin{array}{ll} \tau[\mathit{Monotype}] & ::= \alpha \mid \mathit{Int}^b \mid \tau \rightarrow^b \tau \\ c[\mathit{Constraint}] & ::= b \leq b \mid b \rhd \tau \mid \phi : \tau \leq \tau \end{array}$$

The complete set of binding-time type inference rules can be found in the appendix; here we focus on the rule for application:

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \rightarrow \tau_2)^b \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1\ @^b\ \phi\ e_2) : \tau_2}$$

As usual in a binding-time type system, our judgements depend both on an environment Γ and a set of constraints C. Notice, however, that our subtype constraints include the coercion that maps one type to the other. Thus, from the constraint set C, we infer *which* coercion ϕ converts τ_3 to τ_1. Notice also that we include $\rhd$-constraints to guarantee that the type of the function is well-formed. Finally, the annotation on the application is taken from the type of the function.

Our constraint inference rules, with judgements of the form $C \vdash c$, can be found in the appendix. They are mostly standard, with the exception that the rules for subtyping actually construct a coercion. For example, the rule for function types

$$\frac{C \vdash \phi_1{:}\tau_3 \leq \tau_1 \quad C \vdash \phi_2{:}\tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \rightarrow^{b_1 b_2} \phi_2 : \tau_1 \rightarrow^{b_1} \tau_2 \leq \tau_3 \rightarrow^{b_2} \tau_4}$$

constructs a coercion on functions from coercions on the argument and result. Where possible, we use the identity coercion

$$C \vdash \iota : \tau \leq \tau$$

which can be removed altogether by a post-processor. We restrict the coercions in C to be distinct coercion variables; thus we can think of C as a kind of environment, binding coercion variables to their types.

As in the Hindley-Milner type system, **let**-bound variables may have *type schemes* rather than monotypes. Type-schemes take the form

$$\begin{array}{ll} \gamma[\mathit{Qualified\ type}] & ::= \tau \mid q \Rightarrow \gamma \\ q[\mathit{Qualifier}] & ::= b \leq b \mid b \rhd \tau \mid \tau \leq \tau \\ \pi[\mathit{Polychronic\ type}] & ::= \gamma \mid \forall \beta . \pi \\ \sigma[\mathit{Polymorphic\ type}] & ::= \pi \mid \forall \alpha . \sigma \end{array}$$

We give a complete set of rules to introduce and eliminate type-schemes in the appendix; note that although our rule system is not syntax-directed, it is easy to transform it into a syntax-directed system because of the restriction on where type schemes may appear. Here we discuss only the rules which are not standard.

Notice that qualifiers are almost, but not exactly, the same as constraints. The difference is that sub-type qualifiers $\tau_1 \leq \tau_2$ do not mention a coercion. Looking at the rules for introducing and eliminating such a qualifier

$$\frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma} \qquad \frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

we see why: the coercion in the constraint becomes the bound variable of a coercion abstraction; it would be unnatural to allow bound variable names in types. That we 'forget' the coercion doesn't matter: it can be recreated where it is needed by the elimination rule.

The rules for generalising and instantiating type variables are standard, except that we only allow instantiation with well-formed types. The rules for binding-time variables just introduce binding-time abstraction and application:

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

Given any unannotated expression which is well-typed in the Hindley-Milner system, we can construct a well-typed annotated expression by annotating each application with a fresh binding-time variable and a fresh coercion variable, moving constraints into qualified types, and generalising all possible variables. But this leads to polymorphic definitions with very many generalised variables, and very many qualifiers. In the remainder of this section we will see how to reduce this multitude.

3.2 Simplifying Constraints

Before generalising the type of a **let**-bound variable, it is natural to simplify the constraints as much as possible. Simplification of this kind of constraint is mostly standard [12], except that we keep track of coercions also; essentially we use the constraint inference rules in the appendix backwards, instantiating variables where necessary to make rules match. For example, we simplify the constraint $\xi : \alpha \leq \mathit{Int}^b$ by instantiating α to Int^β and ξ to $\mathit{Int}^{\beta b}$, where β is fresh, and then simplifying the constraint to $\beta \leq b$. Simplification of this kind does not change the set of solutions of the constraints.

We use two non-standard simplification rules also. Firstly, whenever we discover a cycle of binding-time variables $\beta_1 \leq \cdots \leq \beta_n \leq \beta_1$, we instantiate each β_i to the same variable. We treat cycles of type variables similarly, which much reduces the number of variables we need to quantify over. Secondly, we simplify the constraints $\{D \rhd \alpha_1, \xi : \alpha_1 \leq \alpha_2\}$ by instantiating α_2 to α_1 and ξ to ι: this preserves the set of solutions because both α_1 and α_2 have to be well-formed types annotated D at the top-level, and one such type can be a subtype of another only if they are equal.

Simplification terminates, which can be shown by a lexicographic argument: each rule reduces the size of types, the number of $\rhd$-constraints, the number of $\sqcup$s to the left of $\leq$, or the total number of constraints.

3.3 Simplifying Polymorphic Types

The simplifications in the previous section preserve the set of instances of a polymorphic type. That is, if we simplify a type scheme σ_1 to a type scheme σ_2, then any instance τ_1 of σ_1 is guaranteed also to be an instance of σ_2. But we can go further, if we guarantee only that there is an instance τ_2 of σ_2 which is a *subtype* of τ_1. This still enables us to use a polymorphic value of type σ_2 at any instance of σ_1, provided we introduce a coercion. For example, we can simplify the type of the *power* function from $\forall \beta_1, \beta_2, \beta_3.(\beta_1 \leq \beta_3, \beta_2 \leq \beta_3) \Rightarrow \mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_3}$ to $\forall \beta_1, \beta_2.\mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_1 \sqcup \beta_2}$; these two types do not have the same instances, but any instance of the first can be derived by coercing an instance of the second. The second type has fewer quantified variables and coercions, and is therefore cheaper to specialise.

This subtype condition is guaranteed by ensuring that variables occurring negatively in the type are only instantiated to smaller quantities, while variables occurring positively are only instantiated to larger ones. Moreover, simplification must not increase the binding-time of any program annotation, otherwise it would lead to poorer specialisation. Positively occurring binding-time variables therefore cannot be instantiated at all. Dussart et al. [7] simplify by instantiating non-positive binding-time variables to the least upper bound of their lower bounds (as in the *power* example above).

In the presence of polymorphism, we instantiate type variables also. We might treat non-positive type variables in the same way that Dussart et al. treat binding-time variables, but this would introduce least upper bounds of type variables. This would be problematic for us, since we pass coercions and not types as parameters during specialisation: while it is straightforward (if expensive) to compute the least upper bound of two types, computing the least upper bound of two coercions would be far harder. But in two special cases, we can instantiate non-positive type variables to smaller types *without* needing least upper bounds.

Firstly, if $\xi : \alpha_1 \leq \alpha_2$ is the *only* constraint imposing a lower bound on α_2, and α_2 is non-positive, then we can instantiate α_2 to α_1 and ξ to ι. We also must insist that α_1 and α_2 are forced by the same set of binding-times; otherwise unifying them might make α_1 more dynamic.

Secondly, if α_1 and α_2 are *both* non-positive, have the same set of lower bounds, and are forced by the same binding-times, then they must take the same value in the least solution of the constraints, and we can unify them — even though we cannot express this least solution without least upper bound.

But there is another way to simplify constraints on type variables: we can instantiate non-negative type variables to *larger* types! This does potentially make some *types* more dynamic, but no *binding-times*, and it is the binding-time annotations which determine the quality of specialisation, not the types. We can do this in cases analogous to the two above, except that we need not be concerned with the binding-times which force type variables, since we *expect* to make type variables more dynamic. This process is specified formally in the appendix.

This form of simplification terminates since each step eliminates one variable.

For example, the type inferred for the *map* function, after simplifying binding-times, is

$$\begin{array}{l}\forall \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 . \forall \beta_1, \beta_2 . \\ \quad (\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_3, \beta_2 \rhd \alpha_4, \alpha_3 \leq \alpha_1, \alpha_2 \leq \alpha_4, \alpha_5 \leq \alpha_4) \Rightarrow \\ \quad\; (\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{S} [\alpha_3]^{\beta_2} \rightarrow^{S} [\alpha_4]^{\beta_2}\end{array}$$

(where α_5 is a type variable internal to the definition of *map*). α_4 has two lower bounds, so cannot be reduced, while α_1 cannot be reduced to its only lower bound α_3 since $\beta_1 \rhd \alpha_1$, but β_1 does not force α_3. However, α_2, α_3, and α_5 are all non-positive and have unique upper bounds, so we can increase all three to their upper bounds and simplify the type to

$$\begin{array}{l}\forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_1, \beta_2 \rhd \alpha_2) \Rightarrow \\ \quad (\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{S} [\alpha_1]^{\beta_2} \rightarrow^{S} [\alpha_2]^{\beta_2}\end{array}$$

The number of coercion parameters is decreased from three to zero.

4 Handling Recursion

We have implemented this binding-time analysis in a prototype partial evaluator for polymorphic programs [11]. Handling recursive programs, in particular, forced us to introduce hand annotations which differ from those required in a monomorphic setting.

Consider the annotated *power* function:

$$\begin{aligned}
\mathit{power}\ \beta_1\ \beta_2\ n\ x = {} & \mathbf{if}^{\beta_1}\ n =^{\beta_1} \mathit{Int}^{S\beta_1}\ 0 \\
& \mathbf{then}\ \mathit{Int}^{S(\beta_1 \sqcup \beta_2)}\ 1 \\
& \mathbf{else}\ x \times^{\beta_1 \sqcup \beta_2} \mathit{power}\ \beta_1\ \beta_2 @^S (n -^{\beta_1} \mathit{Int}^{S\beta_1}\ 1) @^S x
\end{aligned}$$

The specialiser unfolds function calls, so specialising this definition to a static value for n produces a compact expression — for example,

$$\mathit{power}\ S\ D\ 3\ \text{“}x\text{”} = \text{“}x \times x \times x \times 1\text{”}$$

But if n is dynamic, then unfolding would produce an infinite residual program:

$$\begin{aligned}
\mathit{power}\ D\ S\ \text{“}n\text{”}\ 2 = \text{“} & \mathbf{if}\ n = 0 \\
& \mathbf{then}\ 1 \\
& \mathbf{else}\ \mathbf{if}\ n - 1 = 0 \\
& \qquad \mathbf{then}\ 2 \\
& \qquad \mathbf{else}\ \ldots\text{”}
\end{aligned}$$

The standard solution is to generate a recursive residual program instead:

$$\begin{aligned}
\mathit{power}\ D\ S\ \text{“}n\text{”}\ 2 = \text{“} & \mathit{power}_2\ n \\
& \mathbf{where}\ \mathit{power}_2\ n = \mathbf{if}\ n = 0 \\
& \qquad\qquad\qquad\quad \mathbf{then}\ 1 \\
& \qquad\qquad\qquad\quad \mathbf{else}\ 2 * \mathit{power}_2\ (n-1)\text{”}
\end{aligned}$$

Annotated expressions which are specialised to create residual function definitions are called *program points*, but various strategies have been used in the literature to decide which expressions to treat as program points. We have chosen to indicate program points explicitly via a hand-annotation **pp**. Thus we write the annotated *power* function as

$$\begin{aligned}
\mathit{power}\ \beta_1\ \beta_2\ n\ x = \mathbf{pp}^{\beta_1}\ & \mathbf{if}^{\beta_1}\ n =^{\beta_1} \mathit{Int}^{S\beta_1}\ 0 \\
& \mathbf{then}\ \mathit{Int}^{S(\beta_1 \sqcup \beta_2)}\ 1 \\
& \mathbf{else}\ x \times^{\beta_1 \sqcup \beta_2} \mathit{power}\ \beta_1\ \beta_2 @^S (n -^{\beta_1} \mathit{Int}^{S\beta_1}\ 1) @^S x
\end{aligned}$$

In the standard (monomorphic) setting, specialising a program point always creates a new residual definition; program points are never unfolded. In our setting, when binding-times are not fixed in advance, this would be too inflexible. Whether the program point in this example should be unfolded depends on the binding-time of n, which is β_1. Thus we annotate **pp** constructions with a binding-time, which indicates whether a residual function definition should be created or not.

However, the programmer does not write annotated programs. He writes a source program, which is transformed into an annotated one by the binding-time analyzer. Since the programmer cannot write binding-times directly, we need to provide a different way to control the binding-time which is attached to a **pp**. Our solution is to give **pp** an extra argument in the source language: an expression whose top-level binding-time is used to annotate the program point. In this example, since the program point is controlled by the binding-time of n, we write

```
power n x = pp n (if n = 0
                  then 1
                  else x × power (n − 1) x)
```

in the source language. If n is static, then the specialiser unfolds this definition as before.

In fact, we use CPS-style specialisation, which moves static contexts into the branches of residual conditionals. For an example, see the "infinite" specialisation of *power* above, where the static $2\times$ was moved into the branches of the generated conditionals, where it could be performed during specialisation. Our specialiser also moves static contexts into specialised program points, with the pleasant consequence that inserting a **pp** annotation does not change any binding-times. However, in the *power* example this leads the specialiser to generate infinitely many residual functions: one where the result is 1, one where it is 2, one where it is 4, etc. To prevent this happening, we must *force* the binding-time of the result to be D, if n is dynamic. In practice, every binding-time analyser sometimes makes *too many* operations static, causing partial evaluation to loop, and this must be prevented using user annotations [15]. But in our context, whether we should force an expression to be dynamic or not depends on other binding-times.

We therefore add another annotation to the source language, **force** e_1 e_2. This evaluates to e_2, but is forced to be dynamic if e_1 is dynamic. The binding-time analyser transforms this into a suitable coercion; **force** does not appear in annotated programs, it only steers the result of BTA. Using **force**, we can write the *power* function as

```
power n x = pp n (if n = 0
                  then 1
                  else force n x × power (n − 1) x)
```

which is then specialised as in the examples above.

These annotations force a *binding-time* to be dynamic, if a *type* is dynamic. We introduce a new form of constraint, $\tau \rhd b$, to express this. (Previously we used constraints between types and types, binding-times and types, and binding-times and binding-times, but we had no way to constrain a binding-time based on a type). This constraint is satisfied if the top-level binding-time of τ is less than b:

$$\frac{C \vdash b_1 \le b_2}{C \vdash \mathit{Int}^{b_1} \rhd b_2} \qquad \frac{C \vdash b_1 \le b_2}{C \vdash \tau_1 \rightarrow^{b_1} \tau_2 \rhd b_2}$$

Our implementation uses these rules to simplify constraints of the new form, but is not yet able to solve constraints $\alpha \rhd b$ involving a type variable, if any remain after simplification. We do not expect any fundamental difficulties in doing so, but for now we accept the mild restriction that a **pp** or **force** annotation in a polymorphic function definition may not be controlled by an expression whose type is just a type variable. This restriction guarantees that all such constraints can be simplified using the rules above, and has not proved awkward in practice.

5 Discussion

Polymorphism is particularly important for programs made up of many modules. In earlier work on specialising modules [9,6,10] we discovered we needed polymorphic binding-time analysis, which directly inspired this work.

Our analysis is built on Henglein et al's earlier polychronic analyses [13, 7]. Our extensions treat polymorphic types, introduce CPS specialisation, and consider hand annotations necessary to control the BTA. Consel et al developed a polychronic binding time analysis based on types and effects, similar in spirit to Henglein et al's [4]. Binding-time analysers for polymorphic programs have also been developed based on abstract interpretation [17,21], although this approach is now little used in practice.

In parallel with our own work, Glynn et al. generalised Henglein's BTA to polymorphic programs, representing constraints by boolean formulæ. This enables them to use a standard BDD package for efficient constraint solving. However, they have not implemented a corresponding specialiser (which tends to have an impact on the BTA in turn), and their analysis is currently applicable only to non-strict languages [8].

This paper considers only parametric polymorphism, without overloading. We hope to extend our system to handle overloading based on Haskell classes [23]. Another important extension is to handle recursive datatypes: we considered these in a slightly different context in [10].

A much more detailed description of this work, including a complete specification of the associated partial evaluator, can be found in Heldal's thesis [11].

6 Conclusion

Polymorphic typing is integral to widely used functional programming languages such as ML and Haskell, and has also been adopted in other languages such as Mercury [22]. Polymorphic binding-time analysis, such as ours, is vital if program specialisation is to be applied to such languages in practice.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. A. Bondorf. Automatic autoprojection of higher-order recursion equations. In N. Jones, editor, *3rd European Symposium on Programming*, LNCS, Copenhagen, 1990. Springer-Verlag.
3. A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, June 1992.
4. C. Consel and P. Jouvelot. Separate Polyvariant Binding-Time Analysis. Technical Report CS/E 93-006, Oregon Graduate Institute Tech, 1993.

5. Charles Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, June 1993.
6. D. Dussart, R. Heldal, and J. Hughes. Module-Sensitive Program Specialisation. In *Conference on Programming Language Design and Implementation*, Las Vegas, June 1997. ACM SIGPLAN.
7. D. Dussart, F. Henglein, and C. Mossin. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In Alan Mycroft, editor, *SAS'95: 2nd Int'l Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.
8. Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Söndergård. Boolean constraints for binding-time analysis. In Olivier Danvy and Andrzej Filinski, editors, *Second Symposium on Programs as Data Objects*, volume 2053 of *LNCS*, pages 39–62, Aarhus, May 2001. Springer-Verlag.
9. R. Heldal and J. Hughes. Partial Evaluation and Separate Compilation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, June 1997. ACM SIGPLAN.
10. R. Heldal and J. Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science 248*, pages 99–145, 2000.
11. Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator.* PhD thesis, Chalmers University of Technology, April 2001.
12. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.
13. F. Henglein and C. Mossin. Polymorphic Binding-Time Analysis. In D. Sannella, editor, *ESOP'94: European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.
14. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
15. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
16. Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
17. J. Launchbury. *Projection Factorisations in Partial Evaluation (PhD thesis)*, volume 1 of *Distinguished Dissertations in Computer Science.* Cambridge University Press, 1991.
18. K. Malmkjær, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *Workshop on ML and its Applications*, pages 112–119. ACM SIGPLAN, 1994.
19. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.
20. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

21. T. Æ. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In *Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, March 1989.
22. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
23. P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

A Appendix: Binding-Time Rules

$$\Gamma, x{:}\sigma, \Gamma'; C \vdash x : \sigma \qquad \Gamma; C \vdash c : Int^S \qquad \frac{C \vdash \tau_1 \text{ wft} \quad \Gamma, x{:}\tau_1; C \vdash e : \tau_2}{\Gamma; C \vdash \lambda x.e : (\tau_1 \to^S \tau_2)}$$

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \to \tau_2)^b \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1 \mathbin{@^b} \phi\ e_2) : \tau_2}$$

$$\frac{\Gamma; C \vdash e_1 : \sigma \quad \Gamma, x{:}\sigma; C \vdash e_2 : \tau}{\Gamma; C \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

Fig. 1. Syntax directed binding-time rules for expressions

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

$$\frac{\Gamma; C, b_1 \leq b_2 \vdash e : \gamma}{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma} \qquad \frac{\Gamma; C, b \rhd \tau \vdash e : \gamma}{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma} \qquad \frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma}$$

$$\frac{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma \quad C \vdash b_1 \leq b_2}{\Gamma; C \vdash e : \gamma} \qquad \frac{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma \quad C \vdash b \rhd \tau}{\Gamma; C \vdash e : \gamma}$$

$$\frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

$$\frac{\Gamma; C \vdash e : \sigma}{\Gamma; C \vdash e : \forall\alpha.\sigma}\ \alpha \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\alpha.\sigma \quad C \vdash \tau \text{ wft}}{\Gamma; C \vdash e : \sigma[\tau/\alpha]}$$

Fig. 2. Non-syntax directed rules

$$C, c \vdash c \qquad C \vdash \iota : \tau \leq \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash \mathit{Int}^{b_1 b_2} : \mathit{Int}^{b_1} \leq \mathit{Int}^{b_2}}$$

$$\frac{C \vdash \phi_1{:}\tau_3 \leq \tau_1 \quad C \vdash \phi_2{:}\tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \rightarrow^{b_1 b_2} \phi_2 : \tau_1 \rightarrow^{b_1} \tau_2 \leq \tau_3 \rightarrow^{b_2} \tau_4}$$

$$\frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd \mathit{Int}^{b_2}} \qquad C \vdash S \rhd \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd \tau_1 \rightarrow^{b_2} \tau_2}$$

$$C \vdash b \leq b \qquad C \vdash S \leq b \qquad C \vdash b \leq D \qquad C \vdash \beta_i \leq \sqcup \beta_i \qquad \frac{C \vdash \beta_1 \leq \beta_3 \quad C \vdash \beta_2 \leq \beta_3}{C \vdash \beta_1 \sqcup \beta_2 \leq \beta_3}$$

Fig. 3. Constraint inference rules

$$C \vdash \alpha \text{ wft} \qquad C \vdash \mathit{Base}^b \text{ wft} \qquad \frac{C \vdash \tau_1 \text{ wft} \quad C \vdash \tau_2 \text{ wft} \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{C \vdash \tau_1 \rightarrow^b \tau_2 \text{ wft}}$$

Fig. 4. Well-formedness of types

Each time one of the rules below is applied, the constraints must first be normalised and the set of force constraints must be closed using the following rule:

$$\{\beta \rhd \alpha_1, \xi : \alpha_1 \leq \alpha_2\} \rightsquigarrow \{\beta \rhd \alpha_1, \beta \rhd \alpha_2, \xi : \alpha_1 \leq \alpha_2\}$$

To simplify a type τ and constraint set C in an environment Γ:

$$\beta \notin (|\tau|^- \cup FV(\Gamma)) \Rightarrow C \rightsquigarrow C_\beta[\beta := \sqcup_{\beta' \in C_{\leq\beta}} \beta']; \beta := \sqcup_{\beta' \in C_{\leq\beta}} \beta'$$

$$\begin{array}{l} \alpha \notin (|\tau|^- \cup FV(\Gamma)) \ \wedge \ C_{\leq\alpha} = \{\} \ \wedge \ C_{\rhd\alpha} \subseteq C_{\rhd\alpha_1} \\ \Rightarrow C, \xi : \alpha_1 \leq \alpha \rightsquigarrow C[\alpha := \alpha_1]; \alpha := \alpha_1, \xi := \iota \end{array}$$

$$\begin{array}{l} \alpha_1, \alpha_2 \notin (|\tau|^- \cup FV(\Gamma)) \ \wedge \ C_{\leq\alpha_1} = C_{\leq\alpha_2} \ \wedge \ C_{\rhd\alpha_1} = C_{\rhd\alpha_2} \\ \Rightarrow C \rightsquigarrow C[\alpha_1 := \alpha_2]; \alpha_1 := \alpha_2 \end{array}$$

$$\begin{array}{l} \alpha \notin (|\tau|^+ \cup FV(\Gamma)) \wedge \ C_{\alpha\leq} = \{\} \\ \Rightarrow C, \xi : \alpha \leq \alpha_1 \rightsquigarrow C[\alpha := \alpha_1]; \alpha := \alpha_1, \xi := \iota \end{array}$$

$$\begin{array}{l} \alpha_1, \alpha_2 \notin (|\tau|^+ \cup FV(\Gamma)) \ \wedge \ C_{\alpha_1\leq} = C_{\alpha_2\leq} \\ \Rightarrow C \rightsquigarrow C[\alpha_1 := \alpha_2]; \alpha_1 := \alpha_2 \end{array}$$

where

$$\begin{array}{lll} C_{\leq\beta} \triangleq \{\beta_1 | \beta_1 \leq \beta \in C\} & C_\beta \triangleq C - \{\beta_1 \leq \beta \ | \beta_1 \in \mathbb{B}\} & \\ C_{\leq\alpha} \triangleq \{\alpha_1 | \xi : \alpha_1 \leq \alpha \in C\} & C_{\rhd\alpha} \triangleq \{b | b \rhd \alpha \in C\} & C_{\alpha\leq} \triangleq \{\alpha_1 | \xi : \alpha \leq \alpha_1 \in C\} \end{array}$$

Fig. 5. Increasing and decreasing variables

An Investigation of Compact and Efficient Number Representations in the Pure Lambda Calculus

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
phone: (+45) 35321404 fax: (+45) 35321401 `torbenm@diku.dk`

Abstract. We argue that a compact right-associated binary number representation gives simpler operators and better efficiency than the left-associated binary number representation proposed by den Hoed and investigated by Goldberg. This representation is then generalised to higher number-bases and it is argued that bases between 3 and 5 can give higher efficiency than binary representation.

1 Introduction

The archetypal number representation in the pure lamda calculus is Church numerals, where a number n is represented as a combinator that applies a function n times to its argument, so for example 3 is represented as $\lambda fx.f\ (f\ (f\ x))$. The definitions of addition, multiplication and raising to power are extremely simple when using Church numerals. However, Church numerals are not very compact and the operations, though simple, are quite costly. den Hoed [6] suggested a compact representation of binary numbers, where a binary number $b_n \dots b_1 b_0$ is represented as $\lambda x_0 x_1.x_{b_0}\ x_{b_1} \dots x_{b_n}$ or, equivalently, $\lambda x_0 x_1.((x_{b_0}\ x_{b_1}) \dots x_{b_n})$. This representation is, hence, *left-associated*.

Given this as a challenge, Mayer Goldberg [2] has shown that efficient operators exist for this representation. However, we believe we can achieve better by using a right-associated representation and introducing one more variable to mark the end of a bit sequence: 0 is represented by $\lambda z x_0 x_1.z$ and $b_n \dots b_1 b_0$, where $b_n \neq 0$ is represented by $\lambda z x_0 x_1.x_{b_0}\ (x_{b_1}\ (\dots (x_{b_n}\ z))$.

We use standard lambda calculus notation: Identity function: $I \equiv \lambda x.x$, booleans: $T \equiv \lambda xy.x$, $F \equiv \lambda xy.y$, tuples: $[e_1, \dots, e_n] \equiv \lambda x.x\ e_1\ \dots\ e_n$ and projections: $\pi^n_k \equiv \lambda t.t(\lambda x_1 \cdots x_n.x_k)$. The identity function is $I \equiv \lambda x.x$. We use the notation $\lceil n \rceil$ to mean the representation of the number n. Examples:

$$\begin{aligned}\lceil 0 \rceil &\equiv \lambda z x_0 x_1.z\\ \lceil 1 \rceil &\equiv \lambda z x_0 x_1.x_1\ z\\ \lceil 5 \rceil &\equiv \lambda z x_0 x_1.x_1\ (x_0\ (x_1\ z))\end{aligned}$$

We note that this representation, like most binary representations, allow leading zeroes. For example, 0 can be represented as $\lambda z x_0 x_1.x_0\ x$ as well as $\lambda z x_0 x_1.z$. We want to avoid introducing leading zeroes. Goldberg [2] builds this into the operators, but we define a separate operator that is applied when needed.

2 Basic Operations on Numbers

Shifting a binary number up by one bit is easily done in constant time:

$$\uparrow \equiv \lambda bn.\lambda z x_0 x_1.b\ x_0\ x_1\ (n\ z\ x_0\ x_1)$$

Single bits are represented as $0 \equiv T$, $1 \equiv F$. For brevity and slightly better efficiency, we will often use specialised versions of $\uparrow$:

$$\uparrow_0 \equiv \lambda n.\lambda z x_0 x_1.x_0\ (n\ z\ x_0\ x_1) \qquad \uparrow_1 \equiv \lambda n.\lambda z x_0 x_1.x_1\ (n\ z\ x_0\ x_1)$$

We can make a function that finds the least significant bit in a number by

$$\mathit{lsb} \equiv \lambda n.n\ T\ (\lambda x.T)\ (\lambda x.F)$$

The idea in this operator is that we apply a number to three terms: One for handling the empty bit string, one for handling the case where the least significant bit is 0 and one for the case where the least significant bit is 1.

We define, in a similar way, operators for stripping leading zeroes and dividing a binary number by 2:

$$\begin{array}{lll} \mathit{strip} & \equiv \lambda n.\pi^2_1\ (n\ Z\ A\ B) & \text{where} \\ Z & \equiv [\ulcorner 0 \urcorner, T] & \\ A & \equiv \lambda p.p\ (\lambda nz.[z\ \ulcorner 0 \urcorner\ (\uparrow_0\ n), z]) & \\ B & \equiv \lambda p.p\ (\lambda nz.[\uparrow_1\ n, F]) & \end{array} \qquad \begin{array}{lll} \downarrow & \equiv \lambda n.\pi^2_2\ (n\ Z\ A\ B) & \text{where} \\ Z & \equiv [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner] & \\ A & \equiv \lambda p.p\ (\lambda nm.[\uparrow_0\ n, n]) & \\ B & \equiv \lambda p.p\ (\lambda nm.[\uparrow_1\ n, n] & \end{array}$$

For the *strip* operator, we build a pair that contains the most compact representation of the bits that have been treated and an indication if this is the empty bit sequence. Finally, we use π^2_1 to extract the final result from the pair. For division, the pair contains the whole number and the number divided by 2.

An adequate number system needs operators for zero-testing, increment and decrement. We make these using the same techniques as above:

$$\mathit{zero?} \equiv \lambda n.n\ T\ I\ (\lambda x.F)$$

$$\begin{array}{lll} \mathit{succ} & \equiv \lambda n.\pi^2_2\ (n\ Z\ A\ B) & \text{where} \\ Z & \equiv [\ulcorner 0 \urcorner, \ulcorner 1 \urcorner] & \\ A & \equiv \lambda p.p\ (\lambda nm.[\uparrow_0\ n, \uparrow_1\ n]) & \\ B & \equiv \lambda p.p\ (\lambda nm.[\uparrow_1\ n, \uparrow_0\ m]) & \end{array} \qquad \begin{array}{lll} \mathit{pred} & \equiv \lambda n.\pi^2_2\ (n\ Z\ A\ B) & \text{where} \\ Z & \equiv [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner] & \\ A & \equiv \lambda p.p\ (\lambda nm.[\uparrow_0\ n, \uparrow_1\ m]) & \\ B & \equiv \lambda p.p\ (\lambda nm.[\uparrow_1\ n, \uparrow_0\ n]) & \end{array}$$

pred can introduce a leading zero if the number is a power of two.

3 Other Number Bases

It is easy to generalise the above representation to other number bases. If we have an n-digit number $d_{n-1} \ldots d_1 d_0$ (where $d_{n-1} > 0$) in base b, we can represent this as $\lambda z x_0 x_1 \ldots x_{n-1}.x_{d_0}\ (x_{d_1}\ (\ldots (x_{d_{n-1}}\ z) \ldots))$. 0 is, of course, represented as $\lambda z x_0 x_1 \ldots x_{n-1}.z$. The operations are also easily generalised:

$$\begin{aligned} \uparrow_i &\equiv \lambda n.\lambda z x_0 \ldots x_{b-1}.x_i\ (n\ z\ x_0\ \ldots\ x_{b-1}) \\ \uparrow\ &\equiv \lambda dn.\lambda z x_0 \ldots x_{b-1}.d\ x_0\ \ldots\ x_{b-1}\ (n\ z\ x_0\ \ldots\ x_{b-1}) \end{aligned}$$

where a digit d for the general $\uparrow$ operator is represented as $\lambda x_0 \ldots x_{b-1}.x_d$. The *lsb* operator is replaced by an *lsd* (least significant digit) operator:

$$lsd \equiv \lambda n.n\ (\lambda x_0 \ldots x_{b-1}.x_0)\ (\lambda x.\lambda x_0 \ldots x_{b-1}.x_1)\ \ldots\ (\lambda x.\lambda x_0 \ldots x_{b-1}.x_{b-1})$$

Digits are represented as above. The strip operator is simple to generalise, as are diving by the base (*i.e.*, removing the last digit), the zero test and the successor and predecessor operators:

$$\begin{aligned} strip &\equiv \lambda n.\pi_1^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) && \text{where} \\ Z &\equiv [\lceil 0 \rceil, T] \\ A_0 &\equiv \lambda p.p\ (\lambda nz.[z\ \lceil 0 \rceil\ (\uparrow_0 n), z]) \\ A_i &\equiv \lambda p.p\ (\lambda nz.[\uparrow_i n, F])\ \ , \quad i > 0 \\[1em] \downarrow\ &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) && \text{where} \\ Z &\equiv [\lceil 0 \rceil, \lceil 0 \rceil] \\ A_i &\equiv \lambda p.p\ (\lambda nm.[\uparrow_i n, n]) \\[1em] zero? &\equiv \lambda n.n\ T\ I\ (\lambda x.F)^{b-1} \\[1em] succ &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) && \text{where} \\ Z &\equiv [\lceil 0 \rceil, \lceil 1 \rceil] \\ A_i &\equiv \lambda p.p\ (\lambda nm.[\uparrow_i n, \uparrow_{i+1} n]) && , i < b-1 \\ A_{b-1} &\equiv \lambda p.p\ (\lambda nm.[\uparrow_{b-1} n, \uparrow_0 m]) \\[1em] pred &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) && \text{where} \\ Z &\equiv [\lceil 0 \rceil, \lceil 0 \rceil] \\ A_0 &\equiv \lambda t.t\ (\lambda nmz.[\uparrow_0 n, \uparrow_{b-1} m] \\ A_i &\equiv \lambda t.t\ (\lambda nmz.[\uparrow_i n, \uparrow_{i-1} n]) && , i > 0 \end{aligned}$$

4 Comparing Different Number Bases

Using he representations shown above, each digit in a base b number is represented by a variable and an application. This could lead us to believe that we can get arbitrarily compact representations by choosing higher number bases. But an actual representation of a lambda term on a machine will have to represent variables using a fixed alphabet, so the number of symbols needed to represent a variable depend on the number of different variables used. In order to be precise about this, we will use de Bruijn notation: each occurrence of a variable is replaced by a so-called de Bruijn number that counts the number of lambda abstractions one has to pass in the syntax tree to get to the abstraction that binds the variable. Abstractions no longer name the bound variable.

We still haven't solved the problem, as we have replaced an unbounded number of variables by an unbounded number of de Bruijn numbers. We can, however, replace these by number strings. A common representation of lambda terms uses unary representation for de Bruijn numbers such that, *e.g.*, 3 is represented as $sssz$. This has the operational interpretation that z returns the value of the variable at the front of the current environment while s walks down one level in the environment. This idea is the basis of several abstract machines [1] [3].

We will explicitly bracket applications. By omitting the (redundant) opening bracket, we get a representation similar to reverse polish notation. Examples:

lambda term	de Bruijn notation	compact representation
$\lambda x.x$	$\lambda 0$	λz
$\lambda xy.y\ x$	$\lambda\lambda 0\ 1$	$\lambda\lambda zsz)$
$\lambda z x_0 x_1.x_1\ (x_0\ (x_1\ z))$	$\lambda\lambda\lambda 0\ (1\ (0\ 2))$	$\lambda\lambda\lambda zszzssz)))$

Using this representation, an n-bit number is represented by a string of length $2.5 \times n + 1$ on average for den Hoed's left-associated representation. The right-associated representation requires and $2.5 \times n + 6$ symbols on average to represent an n-bit number. Given that a base-b digit and an application takes on average $(b+3)/2$ symbols to write using our notation and that you need $n * ln(2)/ln(b)$ base-b digits to represent an n-bit number, we get the following measures of compactness for our base-b representation:

base	2	3	4	5	6	7	8	9	10
compactness	2.5	1.89	1.75	1.72	1.74	1.78	1.83	1.89	1.96

This table indicates that we will get the asymptotically most compact representation if we use the base 5 representation. Since the above compactness measures depend somewhat on the exact details of the textual representation, we must take the numbers with a grain of salt. It is our guess that, for most reasonable measures, it will be better to use a base between 3 and 6 rather than base 2. A rough estimate is that the cost of processing a digit is roughly proportional to the number of different digits, so computational cost should follow a pattern similar to the compactness measures. If the cost per digit is $kb + l$, the cost of processing an n-bit number (in base b) is $(kb+l)ln(2)/ln(b)$. Hence, the minimum is obtained when $b(ln(b) - 1) = l/k$. This gives:

l/k	0	1/4	1/2	2/3	1	3/2	2	3	4
minimum	2.72	2.95	3.18	3.32	3.59	3.97	4.32	4.97	5.57
optimal base	3	3	3	3	4	4	4	5	6

As can be seen, the minimum is always at b higher than 2. This indicates that it will always be better to use base 3 instead of base 2, and it may be better to choose an even higher base. Note that these measures are about asymptotic costs. For small numbers it will be better to use a small number base, such as binary or even unary.

5 Balanced Ternary Representation

An interesting number representation is balanced ternary. This variant base-3 notation has digits "-" (-1), "0" and "+" (1). This allows representation of negative numbers without a separate sign symbol. Balanced ternary has been used in some early Russian computers, but lost to the simplicity of binary electronics. It may, however, well be good for number representation in the lamda calculus.

Representing numbers in balanced ternary is very similar to ordinary ternary representation: We represent a balanced ternary digit string $t_n \dots t_0$, where $t_n \neq 0$, as $\lambda z x_- x_0 x_+ . x_{t_0}\ (\dots (x_{t_n}\ z))$. Examples:

$$\begin{array}{ll}
\lceil 0 \rceil & \equiv \lambda z x_- x_0 x_+ . z \\
\lceil 1 \rceil & \equiv \lambda z x_- x_0 x_+ . x_+\ z \\
\lceil -1 \rceil & \equiv \lambda z x_- x_0 x_+ . x_-\ z \\
\lceil 2 \rceil & \equiv \lambda z x_- x_0 x_+ . x_-\ (x_+\ z) \\
\lceil -5 \rceil & \equiv \lambda z x_- x_0 x_+ . x_+\ (x_+\ (x_-\ z))
\end{array}$$

The operators are similar to those of normal ternary.

$$\begin{array}{ll}
\uparrow_- & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_-\ (n\ z\ x_-\ x_0\ x_+) \\
\uparrow_0 & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_0\ (n\ z\ x_-\ x_0\ x_+) \\
\uparrow_+ & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_+\ (n\ z\ x_-\ x_0\ x_+) \\
\uparrow & \equiv \lambda t n . \lambda z x_- x_0 x_+ . t\ x_-\ x_0\ x_+\ (n\ z\ x_-\ x_0\ x_+)
\end{array}$$

where a ternary digit (trit) is represented as $\hat{-} \equiv \lambda x_- x_0 x_+ . x_-$, $\hat{0} \equiv \lambda x_- x_0 x_+ . x_0$ and $\hat{+} \equiv \lambda x_- x_0 x_+ . x_+$. Operators for least significant trit, zero-testing, addition and subtraction are almost the same as for "ordinary" ternary numbers:

$$
\begin{array}{ll}
lst & \equiv \lambda n.n\ \hat{0}\ (\lambda x.\hat{-})\ (\lambda x.\hat{0})\ (\lambda x.\hat{+}) \\
\\
zero? & \equiv \lambda n.n\ T\ (\lambda x.F)\ I\ (\lambda x.F) \\
\\
succ & \equiv \lambda n.\pi_2^2\ (n\ Z\ A_-\ A_0\ A_+) \qquad \text{where} \\
Z & \equiv [\lceil 0 \rceil, \lceil 1 \rceil] \\
A_- & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_-\ n, \uparrow_0\ n]) \\
A_0 & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_0\ n, \uparrow_+\ n]) \\
A_+ & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_+\ n, \uparrow_-\ m]) \\
\\
pred & \equiv \lambda n.\pi_2^2\ (n\ Z\ A_-\ A_0\ A_+) \qquad \text{where} \\
Z & \equiv [\lceil 0 \rceil, \lceil -1 \rceil] \\
A_- & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_-\ n, \uparrow_+\ m]) \\
A_0 & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_0\ n, \uparrow_-\ n]) \\
A_+ & \equiv \lambda t.t\ (\lambda nmz.[\uparrow_+\ n, \uparrow_0\ n])
\end{array}
$$

Note the symmetry of *succ* and *pred*. They can now both introduce a leading zero. Since we now have negative numbers, an useful operator is negation. This is just a matter of replacing $+$ by $-$ and *vice versa*:

$$negate \equiv \lambda n.\lambda z x_- x_0 x_+.n\ z\ x_+\ x_0\ x_-$$

6 Binary Operators

Some binary operators, like addition, require walking down two digit strings simultaneously. The representations we have shown so far aren't geared to this, so we introduce yet another representation, which allows us to inspect one ternary digit at a time. The representation is similar to the previously shown, except that the variables z, x_-, x_0 and x_+ are abstracted at every digit rather than globally for all digits:

$$
\begin{array}{l}
\lfloor 0 \rfloor \ \equiv\ \lfloor \epsilon \rfloor \equiv \lambda z x_- x_0 x_+.z \\
\lfloor t_n \dots t_1 t_0 \rfloor \equiv \lambda z x_- x_0 x_+.x_{t_0}\ (\lfloor t_n \dots t_1 \rfloor) \qquad , t_n \neq 0
\end{array}
$$

where ϵ represents the empty digit string. We introduce operators $\Uparrow$, which takes a number n in the "normal" balanced ternary representation ($\lceil n \rceil$) to the new representation ($\lfloor n \rfloor$) and its inverse $\Uparrow$:

$$
\begin{array}{ll}
\Uparrow & \equiv \lambda n.n\ \lfloor 0 \rfloor\ A_-\ A_0\ A_+ \qquad \text{where} \\
A_- & \equiv \lambda n.\lambda z x_- x_0 x_+.x_-\ n \\
A_0 & \equiv \lambda n.\lambda z x_- x_0 x_+.x_0\ n \\
A_+ & \equiv \lambda n.\lambda z x_- x_0 x_+.x_+\ n
\end{array}
$$

$$\Downarrow\ n \equiv n\ \lceil 0 \rceil\ (\lambda m.\uparrow_-(\Downarrow\ m))\ (\lambda m.\uparrow_0(\Downarrow\ m))\ (\lambda m.\uparrow_+(\Downarrow\ m))$$

The latter is given by a recursive equation. Solving this by using a recursion operator will make termination dependent on evaluation order. Hence, we delay the recursion until the right-hand side has been reduced. A similar trick was used in [4]:

$$\begin{array}{l} \Downarrow\ \equiv U\ U \qquad \text{where} \\ U \equiv \lambda un.n\ (\lambda u.\lceil 0 \rceil)\ (\lambda um.\uparrow_{-}(u\ u\ m))\ (\lambda um.\uparrow_{0}(u\ u\ m))\ (\lambda um.\uparrow_{+}(u\ u\ m))\ u \end{array}$$

The idea used in the following is that one of the arguments to a binary operator is converted to the new representation by $\Uparrow$ while the other is processed directly in the original representation. Hence, we define an operator eq_0 that takes a number in the original representation and a number in the new representation and compares these for equality. We then define an equality operator $eq \equiv \lambda xy.eq_0\ x\ (\Uparrow y)$ that takes both arguments in the original representation.

$$\begin{array}{lll} eq_0 & \equiv \lambda n.n\ Z\ A_{-}\ A_0\ A_{+} & \text{where} \\ Z & \equiv \lambda n.zero?\ (\Downarrow\ n) & \\ A_{-} & \equiv \lambda en.n\ F\ (\lambda m.e\ m)\ (\lambda m.F)\ (\lambda m.F) & \\ A_0 & \equiv \lambda en.n\ (e\ n)\ (\lambda m.F)\ (\lambda m.e\ m)\ (\lambda m.F) & \\ A_{+} & \equiv \lambda en.n\ F\ (\lambda m.F)\ (\lambda m.F)\ (\lambda m.e\ m) & \end{array}$$

We, similarly, first define an addition operator add_0 that takes its second argument in the new representation and use this to define $add \equiv \lambda xy.add_0\ x\ (\Uparrow y)$. At each step in the addition process, we have two trits and a carry (which is also a trit, represented as $\hat{-}$, $\hat{0}$ or $\hat{+}$). This can give results from -3 to 3, which are output as a trit and a carry, which is used for the next step. We find the trits in one number by applying suitable Z, A_{-}, A_0 and A_{+} to the number. The trits of the other number are obtained by a 4-way branch in each A-term, as in eq_0. The value of the carry is found by a 3-way branch (in each B-term below).

$$\begin{array}{lll} add_0 & \equiv \lambda n.C\ (n\ Z\ A_{-}\ A_0\ A_{+}) & \text{where} \\ Z & \equiv \lambda nc.c\ (pred\ (\Downarrow\ n))\ (\Downarrow\ n)\ (succ\ (\Downarrow\ n)) & \\ A_{-} & \equiv \lambda anc.n\ (B_{-}\ n)\ B_{-2}\ B_{-}\ B_0 & \\ A_0 & \equiv \lambda anc.n\ (B_0\ n)\ B_{-}\ B_0\ B_{+} & \\ A_{+} & \equiv \lambda anc.n\ (B_{+}\ n)\ B_0\ B_{+}\ B_{+2} & \\ B_{-2} & \equiv \lambda m.c\ (\uparrow_0\ (a\ m\ \hat{-}))\ (\uparrow_{+}\ (a\ m\ \hat{-}))\ (\uparrow_{-}\ (a\ m\ \hat{0})) & \\ B_{-} & \equiv \lambda m.c\ (\uparrow_{+}\ (a\ m\ \hat{-}))\ (\uparrow_{-}\ (a\ m\ \hat{0}))\ (\uparrow_0\ (a\ m\ \hat{0})) & \\ B_0 & \equiv \lambda m.\uparrow c\ (a\ m\ \hat{0}) & \\ B_{+} & \equiv \lambda m.c\ (\uparrow_0\ (a\ m\ \hat{0}))\ (\uparrow_{+}\ (a\ m\ \hat{0}))\ (\uparrow_{-}\ (a\ m\ \hat{+})) & \\ B_{+2} & \equiv \lambda m.c\ (\uparrow_{+}\ (a\ m\ \hat{0}))\ (\uparrow_{-}\ (a\ m\ \hat{+}))\ (\uparrow_0\ (a\ m\ \hat{+})) & \\ C & \equiv \lambda an.a\ n\ \hat{0} & \end{array}$$

We have here been a bit sloppy, as the B terms have free variables a and c which correspond to different bound variables depending on where the B is substituted. Subtraction is easily obtained by negating one argument before addition, so we just define the subtraction operator as $sub \equiv \lambda xy.add\ x\ (negate\ y)$. Multiplication is simple to define using addition and subtraction:

$$\begin{array}{lll} mul \equiv \lambda nm.n\ \lceil 0 \rceil\ A_-\ A_0\ A_+ & & \text{where} \\ A_- \equiv \lambda n.sub\ (\uparrow_0\ n)\ m \\ A_0 \equiv \lambda n.(\uparrow_0\ n) \\ A_+ \equiv \lambda n.add\ (\uparrow_0\ n)\ m \end{array}$$

7 Benchmarks

Our claims of efficiency above have, with the exception of the measure of compactness, been rather weakly argued on grounds of simpler operators. To remedy this, we have timed some calculations using different representations. To execute the calculations, we need a normaliser for the lambda calculus. We have elected to use a normaliser based on normalisation by evaluation and implemented in scheme [5]. This uses a call-by-value reduction strategy.

The, admittedly simplistic, test we use is counting from 0 to 50000 using the *succ* operator. While this may not be representative of "real" calculations, we have in the majority of representations only implemented unary operators, which limits the scope of tests we can make on these.

Representation	time to count to 50000
Right-associated binary	6270 ms
Base 3	4380 ms
Base 4	4000 ms
Base 5	3900 ms
Base 6	4470 ms
Balanced ternary	4660 ms

The time used to execute the benchmark drops by more than 30% from binary to base 3, but the advantage of further going to base 4 or 5 is less (around 10%).

This supports the conjecture that the optimal base is higher than 2 and likely to be around 4 or 5. Balanced ternary is slightly slower than ordinary ternary, but that should be no surprise since the benchmark doesn't use negative numbers.

8 Conclusion

We have investigated a number of different compact number representations for the lambda calculus, starting with the left-associated binary number system suggested by den Hoed. We argued that we get better calculation efficiency by choosing a right-associated representation and adding an explicit end symbol. We then found that number bases in the range 3-6 increase compactness and calculation efficiency over binary representation.

While execution efficiency seems optimal at base 4 or 5, the operators become much bigger in these bases than in ternary: The size of the *succ* operator is approximately quadratic in the number base, and the size of the addition operator is approximately cubic in the number base. This may make base 3 the overall best choice. If ease of conversion to/from binary notation is important, a base-4 representation might be preferable.

The ease of handling negative numbers leads us to suggest using a balanced ternary number representation, for which we present some binary operators in addition to the unary operators we presented for the other systems.

While we, arguably, gain efficiency over Goldbergs operators for den Hoed's representation, this may not be an entirely fair comparison: After all, Goldbergs work was an answer to a challenge if he could make decent operations for a specific number system that was not designed for that purpose. Hence, he didn't *a priori* have the freedom we have exploited of changing the number system to gain better efficiency.

References

1. Guy Cousineau, Pierre-Louis Curien, Michel Mauny, and Ascander Suárez. Combinateurs catégorieques et implementation des languages fonctionnels. Technical Report 86-3, LIENS, 1986.
2. Mayer Goldberg. An adequate and efficient left-associated binary numreal system in the λ-calculus. *Journal of Functional Programming*, 10(6):607–623, November 2000.
3. Jean-Louis Krivine. Un interpréteur du λ-calcul. 1985.
4. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.
5. T. Æ. Mogensen. Gödelisation in the untyped lambda calculu. In O. Danvy, editor, *Proceedings of PEPM'99*. BRICS Notes Series, 1999.
6. W. L. van den Poel, C. E. Schaap, and G. van der Mey. New arithmetical operators in the theory of combinators. *Indagationes Mathematicae*, (42):271–325, 1980.

Observational Semantics for Timed Event Structures[⋆]

Irina B. Virbitskaite

A.P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Lavrentjev ave., 630090, Novosibirsk, Russia
virb@iis.nsk.su

Abstract. The paper is contributed to develop a family of observational equivalences for timed true concurrent models. In particular, we introduce three different semantics (based on interleaving, steps, and partial orders of actions) for timed trace and timed bisimulation equivalences in the setting of event structures with dense time domain. We study the relationship between these three approaches and show their discriminating power. Furthermore, when dealing with particular subclasses of the model under consideration such as sequential and deterministic timed event structures there is no difference between a more concrete or a more abstract approach.

1 Introduction

An important ingredient of every theory of concurrency is a notion of equivalence between processes. Over the past several years, a variety of equivalences have been promoted, and the relationship between them has been quite well-understood (see, for example, [7,8]). Two main lines which have been followed can be sketched as follows. The first aspect which is most dominant in the classical concurrency approaches is the so-called linear time – branching time spectrum. Here different possibilities are discussed to what extent the points of choice between different executions of systems are taken into account. In the linear time approach, the behaviour of a system is represented by the set of its possible executions (*trace equivalence* [11]), i.e. points of choice are ignored. At the other end of the spectrum, *bisimulation equivalence* [10,14] considers choices very precisely. The other aspect to follow is whether partial orders between of action occurrences are taken into account. In the interleaving approach, these are neglected. Using more expressive system models like Petri nets or event structures, partial order based equivalences can be easily defined.

Those equivalences were considered for formal system models without time delays. Recently, a growing interest can be observed in modelling real-time systems which imply a need of a representation of the lapse of time. Several formal

⋆ This work is partially supported by the Russian Foundation for Basic Research (Grant N 00-01-00898).

methods for specifying and reasoning about such systems have been proposed in the last ten years (see [2,3] as surveys). Whereas, the incorporation of real time into equivalence notions is less advanced. There are a few papers (see, for example, [6,16,19]) where decidability questions of time-sensitive equivalences are investigated. In the above-mentioned studies, real-time systems are represented by timed interleaving models — parallel timer processes or timed automata, containing fictitious time measuring elements called clocks.

In this paper, we seek to develop a framework for observational equivalences in the setting of a timed true concurrent model. In particular, we introduce three different semantics (based on interleaving, steps, and partial orders of actions) for timed trace and timed bisimulation equivalences in the setting of event structures with dense time domain. This allows us to take into account processes' timing behaviour in addition to their degrees of relative concurrency and nondeterminism. We also study the interrelations between these three approaches to the semantics of timed concurrent systems. Furthermore, when dealing with particular subclasses of the model such as sequential and deterministic timed processes there is no difference between a more concrete or a more abstract approach. This line of research is sometimes referred to as comparative concurrency semantics.

There have been several motivations for this work. One has been the papers [1,8,9,15,17] which have developed concurrent variants of different observational equivalences in the setting of event structures. A next origin of this study has been given by a number of papers (see [6,16,19] among others), which have extensively studied time-sensitive equivalence notions for interleaving models. However, to our best knowledge, the literature of timed true concurrent models has hitherto lacked such the equivalences. In this regard, the papers [4,13] are a welcome exception, where different notions of timed testing have been treated in the framework of timed event structures. Finally, another origin has been the paper [5] where equivalences based on step semantics have been investigated for a class of stochastic Petri nets with discrete time.

The rest of the paper is organized as follows. The basic notions concerning timed event structures are introduced in the next section. The definitions of three different semantics (sequences of actions, multisets of actions, partial orders of actions) of timed trace and timed bisimulaton equivalences are given in the following three sections, respectively. In section 6, we establish the interrelations between the equivalence notions in the setting of the model under consideration and its subclasses. Section 7 contains some conclusions and remarks on future works.

2 Timed Event Structures

In this section, we introduce some basic notions and notations concerning timed event structures.

We first recall a notion of an event structure [18]. The main idea behind event structures is to view distributed computations as action occurrences, called events, together with a notion of causality dependency between events (which is reasonably characterized via a partial order). Moreover, in order to model nondeterminism, there is a notion of conflicting (mutually incompatible) events. A labelling function records which action an event corresponds to. Let *Act* be a finite set of actions.

Definition 1 *A* (labelled) event structure *over Act is a 4-tuple* $S = (E, \leq, \#, l)$*, where*

- E *is a countable set of events;*
- $\leq \subseteq E \times E$ *is a partial order (the* causality relation*), satisfying the* principle of finite causes*:* $\forall e \in E \diamond \{e' \in E \mid e' \leq e\}$ *is finite;*
- $\# \subseteq E \times E$ *is a symmetric and irreflexive relation (the* conflict relation*), satisfying the* principle of conflict heredity*:* $\forall e, e', e'' \in E \diamond e \;\#\; e' \leq e'' \;\Rightarrow\; e \;\# e''$*;*
- $l : E \longrightarrow Act$ *is a labelling function.*

For an event structure $S = (E, \leq, \#, l)$, we define $\smile = (E \times E) \setminus (\leq \cup \leq^{-1} \cup \#)$ (the *concurrency relation*); for $e, f \in E$, we let $e \#^1 f \;\Leftrightarrow\; e \# f \wedge (\forall e', f' \in E \diamond e' \leq e \wedge f' \leq f \wedge e' \# f' \Rightarrow e' = e \wedge f' = f)$ (the *immediate conflict*). For $C \subseteq E$, the *restriction* of S to C is defined as $S\lceil C = (C, \leq \cap (C \times C), \# \cap (C \times C), l\mid_C)$. We shall use $\mathcal{O}$ to denote the empty event structure $(\emptyset, \emptyset, \emptyset, \emptyset)$.

Let $C \subseteq E$. Then C is *left-closed* iff $\forall e, e' \in E \diamond e \in C \;\wedge\; e' \leq e \Rightarrow e' \in C$; C is *conflict-free* iff $\forall e,\; e' \in C \diamond \neg(e \;\#\; e')$; C is a *configuration of* S iff C is left-closed and conflict-free. Let $\mathcal{C}(S)$ denote the set of all finite configurations of S.

Next we present a model of timed event structures which are a timed extension of event structures by associating their events with timing constraints that indicate event occurrence times with regard to a global clock. An execution of a timed event structure is a *timed configuration*, that consists of the configuration and the timing function recording a global time moment at which events occur and satisfies some additional requirements.

Before introducing the concept of a timed event structure, we need to define some auxiliary notations. Let $\mathbf{N}$ be the set of natural numbers, and $\mathbf{R}_0^+$ the set of nonnegative real numbers. Define the set of intervals: $Interv = \{[d_1, d_2] \mid d_1, d_1 \in \mathbf{R}_0^+,\; d_1 \leq d_2\}$.

We are now ready to introduce the concept of timed event structures.

Definition 2 *A* (labelled) timed event structure *over Act is a pair* $TS = (S, D)$*, where*

- $S = (E, \leq, \#, l)$ *is a (labelled) event structure over Act and*
- $D : E \longrightarrow Interv$ *is a timing function such that* $e' \leq_S e \Rightarrow \min D(e') \leq \min D(e)$ *and* $\max D(e') \leq \max D(e)$*.*

In a graphic representation of a timed event structure, the corresponding action labels and time intervals are drawn near to events. If no confusion arises, we will often use action labels rather event identities to denote events. The

<-relation is depicted by arcs (omitting those derivable by transitivity), and conflicts are also drawn (omitting those derivable by conflict heredity). Following these conventions, a trivial example of a labelled timed event structure is shown in Fig. 1.

$$\begin{tikzcd}
TS_1 : & {[3,6]} & {[4,7]} \\
 & a : e_1 \arrow[r] & b : e_2 \\
 & & \# \\
 & & b : e_3 \\
 & & {[4,5]}
\end{tikzcd}$$

Fig. 1.

Timed event structures TS and TS' are *isomorphic* (denoted $TS \simeq TS'$), if there exists a bijection $\varphi : E_{TS} \longrightarrow E_{TS'}$ such that $e \leq_{TS} e' \iff \varphi(e) \leq_{TS'} \varphi(e')$, $e \;\#_{TS}\; e' \iff \varphi(e) \;\#_{TS'}\; \varphi(e')$, $l_{TS}(e) = l_{TS'}(\varphi(e))$, and $D_{TS}(e) = D_{TS'}(\varphi(e))$, for all $e, e' \in E_{TS}$.

Definition 3 *Let $TS = (S, D)$ be a timed event structure, $C \in \mathcal{C}(S)$, and $T : C \longrightarrow \mathbf{R}_0^+$. Then $TC = (C, T)$ is a* timed configuration *of TS iff the following conditions hold:*

(i) $\forall\, e \in C \,\diamond\, T(e) \in D(e)$;
(ii) $\forall\, e, e' \in C \,\diamond\, e \leq_{TS} e' \;\Rightarrow\; T(e) \leq T(e')$;
(iii) $\forall\, e \in (E \setminus C) \,\diamond\, (\max\, D(e) \geq T(e')$ *for all* $e' \in C)$ *or*
$(\max\, D(e) \geq T(e')$ *for some* $e' \in C$ *s.t.* $e' \;\#\; e)$.

Informally speaking, a timed configuration consisting of the configuration and the timing function recording a global time moment at which events occur, satisfies the following requirements:

(*i*) an event can occur at a time when its timing constraints are met;
(*ii*) for all events e and e' occurred if e causally precedes e' then e should temporally precede e';
(*iii*) occurrences of events should not temporally prevent other events to occur except the events whose conflicting events have occurred before the events had time to occur.

The *initial timed configuration* of TS is $TC_{TS} = (\emptyset, 0)$. We use $\mathcal{TC}(TS)$ to denote the set of finite timed configurations of TS.

To illustrate the concept, consider the set of the possible timed configurations of the timed event structure TS_1 shown in Fig. 1: $\{(\emptyset, 0), (\{e_1\}, T_1), (\{e_3\}, T_2), (\{e_1, e_3\}, T_3), (\{e_1, e_2\}, T_4) \mid T_1(e_1) \in [3,5]; T_2(e_3) \in [4,5]; T_3(e_1) \in [3,6], T_3(e_3) \in [4,5]; T_4(e_1) \in [3,5], T_4(e_2) \in [4,5], T_4(e_1) \leq T_4(e_2)\}$.

From now on, for $TC_1 = (C_1, T_1), TC_2 = (C_2, T_2) \in \mathcal{TC}(TS)$ we shall write $TC_1 \longrightarrow TC_2$ iff $C_1 \subseteq C_2$, $T_2|_{C_1} = T_1$, and $\forall e \in C_1\; \forall e' \in (C_2 \setminus C_1) \diamond T_1(e) \leq T_2(e')$.

3 Interleaving Semantics

In this section, we define timed trace and timed bisimulation equivalences based on an interleaving observation on timed event structures.

For this purpose we need the following notation. Let $(Act, \mathbf{R}_0^+) = \{(a,d) \mid a \in Act,\ d \in \mathbf{R}_0^+\}$ be the set of *timed actions*.

In the interleaving semantics, a timed event structure progresses through a sequence of timed configurations by occurrences of timed actions. In a timed configuration $TC_1 = (C_1, T_1)$, the *occurrence* of a timed action (a,d) *leads to* a timed configuration $TC_2 = (C_2, T_2)$ (denoted $TC_1 \overset{(a,d)}{\longrightarrow} TC_2$), if $TC_1 \longrightarrow TC_2$, $C_2 \setminus C_1 = \{e\}$, $l(e) = a$, and $T_2(e) = d$. The leading relation is extended to a sequence of timed actions from $(Act, \mathbf{R}_0^+)^*$ as follows: $TC \overset{(a_1,d_1)}{\longrightarrow} \cdots \overset{(a_n,d_n)}{\longrightarrow} TC' \Leftrightarrow TC \overset{(a_1,d_1)\ldots(a_n,d_n)}{\longrightarrow} TC'$. The set $L_{ti}(TS) = \{w \in (Act, \mathbf{R}_0^+)^* \mid TC_{TS} \overset{w}{\longrightarrow} TC$ for some $TC \in \mathcal{TC}(TS)\}$ is the *ti-language* of TS. As an illustration, consider the ti-language of the timed event structure TS_1, shown in Fig. 1: $\{\epsilon$, (a,d_1), (b,d_2), $(a,d_3)(b,d_4)$, $(b,d_5)(a,d_6) \mid d_1, d_3 \in [3,5],\ d_2, d_4, d_5 \in [4,5],\ d_6 \in [4,6],\ d_3 \leq d_4,\ d_5 \leq d_6\}$.

Definition 4 *Let TS and TS' be timed event structures.*

(i) *TS and TS' are* timed interleaving trace equivalent *(denoted $TS \equiv_{ti} TS'$) iff $L_{ti}(TS) = L_{ti}(TS')$,*
i.e., two timed event structures are timed interleaving trace equivalent, if their ti-languages coincide;

(ii) *TS and TS' are* timed interleaving bisimilar *(denoted $TS \underline{\leftrightarrow}_{ti} TS'$) iff there exists a relation $\mathcal{B} \subseteq \mathcal{TC}(TS) \times \mathcal{TC}(TS')$ satisfying the following conditions: $(TC_{TS}, TC_{TS'}) \in \mathcal{B}$ and for all $(TC, TC') \in \mathcal{B}$ it holds:*

(a) if $TC \overset{(a,d)}{\longrightarrow} TC_1$ in TS, then $TC' \overset{(a,d)}{\longrightarrow} TC_1'$ in TS' and $(TC_1, TC_1') \in \mathcal{B}$ for some $TC_1' \in \mathcal{TC}(TS')$,

(b) if $TC' \overset{(a,d)}{\longrightarrow} TC_1'$ in TS', then $TC \overset{(a,d)}{\longrightarrow} TC_1$ in TS and $(TC_1, TC_1') \in \mathcal{B}$ for some $TC_1 \in \mathcal{TC}(TS)$,

i.e., two timed event structures are timed interleaving bisimilar, if there exists a relation between their bisimilar timed configurations, among which the initial ones, such that the timed configurations obtained by occurring timed actions are also timed interleaving bisimilar.

Considering the timed event structures shown in Fig. 2, we get $TS_3 \equiv_{ti} TS_4$, while $TS_2 \not\equiv_{ti} TS_3$, since, for example, $(b,0)(a,0) \in L_{ti}(TS_3)$ and $(b,0)(a,0) \notin L_{ti}(TS_2)$. Further, we have $TS_4 \underline{\leftrightarrow}_{ti} TS_5$ but $TS_3 \underline{\not\leftrightarrow}_{ti} TS_4$, because in the timed configuration of TS_4, containing only the timed action $(a,1)$, the occurrence of a timed action $(b,1)$ is possible, but it is not always the case in TS_3.

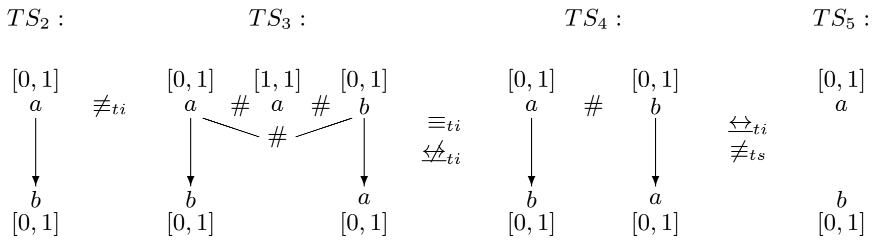


Fig. 2.

4 Step Semantics

In this section, we define a step observation on timed event structures to develop timed step trace and timed step bisimulation equivalences. Step semantics generalizes interleaving semantics by allowing timed actions to occur concurrently with themselves. We show that timed step semantics gives a more precise account of concurrency than the timed interleaving one.

Let A be an arbitrary set. A finite multiset M over A is a function $M : A \longrightarrow \mathbf{N}$ such that $\mid \{a \in A \mid M(a) > 0\} \mid < \infty$. Let $\mathcal{M}^{Act}$ to denote the set of finite nonempty multisets over Act. We use $(\mathcal{M}^{Act}, \mathbf{R}_0^+) = \{(A,d) \mid A \in \mathcal{M}^{Act},\ d \in \mathbf{R}_0^+\}$ to indicate the set of *timed steps*.

In the step semantics, timed configurations of a timed event structure change, if timed steps from $(\mathcal{M}^{Act}, \mathbf{R}_0^+)$ are executed. In a timed configuration $TC_1 = (C_1, T_1)$, the *execution* of a timed step $(A,d) \in (\mathcal{M}^{Act}, \mathbf{R}_0^+)$ *leads* to a timed configuration $TC_2 = (C_2, T_2)$ (denoted $TC_1 \stackrel{(A,d)}{\longrightarrow} TC_2$), if $TC_1 \longrightarrow TC_2,\ C_2 \setminus C_1 = X,\ \forall\ e, e' \in X \diamond e \smile e',\ l(X) = A,\ \forall e \in X \diamond T_2(e) = d$, where $l(X)(a) = |\{e \in X \mid l(e) = a\}|$. The leading relation is extended to a sequence of timed steps from $(\mathcal{M}^{Act}, \mathbf{R}_0^+)^*$ as follows: $TC \stackrel{(A_1,d_1)}{\longrightarrow} \dots \stackrel{(A_n,d_n)}{\longrightarrow} TC' \Leftrightarrow TC \stackrel{(A_1,d_1)\dots(A_n,d_n)}{\longrightarrow} TC'$. The set $L_{ts}(TS) = \{w \in (\mathcal{M}^{Act}, \mathbf{R}_0^+)^* \mid TC_{TS} \stackrel{w}{\longrightarrow} TC$ for some $TC \in \mathcal{TC}(TS)\}$ is the *ts-language* of TS. Considering the timed event structure TS_1, shown in Fig. 1, we have $L_{ts}(TS) = \{\epsilon,\ (\{a\}, d_1),\ (\{b\}, d_2),\ (\{a\}, d_3)(\{b\}, d_4),\ (\{b\}, d_5)(\{a\}, d_6),\ (\{a, b\}, d_2) \mid d_1, d_3 \in [3,5],\ d_2, d_4, d_5 \in [4,5],\ d_6 \in [4,6],\ d_3 \leq d_4,\ d_5 \leq d_6\}$.

Using ts-languages and leading relations of the form $\stackrel{(A,d)}{\longrightarrow}$, we obtain timed step trace equivalence (denoted $\equiv_{ts}$) and timed step bisimulation equivalence (denoted $\underline{\leftrightarrow}_{ts}$) exactly as the corresponding interleaving equivalences in Definition 4. Timed step bisimulation is clearly stronger than timed step trace equivalence.

To illustrate the concepts, consider the timed event structures, shown in Fig. 2 and 3. We have $TS_6 \equiv_{ts} TS_7$, while $TS_4 \not\equiv_{ts} TS_5$, since, for example, $(\{a,b\}, 0) \in L_{ts}(TS_5)$ and $(\{a,b\}, 0) \notin L_{ts}(TS_4)$. Further, we get $TS_7 \underline{\leftrightarrow}_{ts} TS_8$ but $TS_6 \not\underline{\leftrightarrow}_{ts} TS_7$, because in a timed configuration of TS_7, containing only a timed action $(b, 1)$, the execution of the timed step $(\{a\}, 1)$ is always possible, but it is not the case in TS_6.

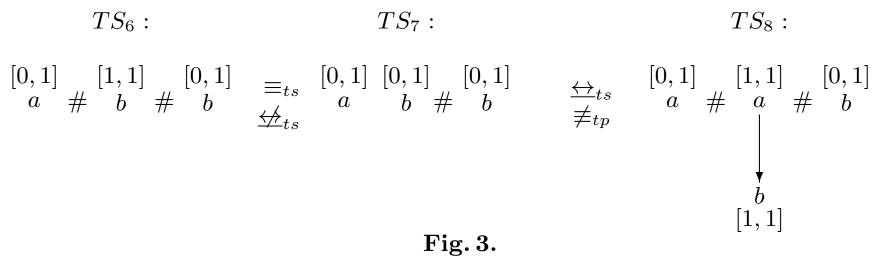

TS_6 :
TS_7 :
TS_8 :
$[0,1]$ a # $[1,1]$ b # $[0,1]$ b
$\equiv_{ts}$
$\not\leftrightarrow_{ts}$
$[0,1]$ a $[0,1]$ b # $[0,1]$ b
$\leftrightarrow_{ts}$
$\not\equiv_{tp}$
$[0,1]$ a # $[1,1]$ a # $[0,1]$ b
b $[1,1]$

Fig. 3.

5 Partial Order Semantics

In this section, we consider several suggestions to define timed equivalence notions based on partial orders which take into account causality between timed actions.

Define a *timed partial order set* as a timed event structure $TP = (S_{TP} = (E_{TP}, \leq_{TP}, \#_{TP}, l_{TP}), D_{TP})$ such that $\#_{TP} = \emptyset$ and $D_{TP} : E_{TP} \longrightarrow Points$, where $Points = \{[d_1, d_2] \in Interv \mid d_1 = d_2\}$. Isomorphism classes of timed partial order sets are called *timed pomsets.*

We now consider leading relations of the form $\xrightarrow{TP}$, where TP is a timed pomset. For $TC_1 = (C_1, T_1), TC_2 = (C_2, T_2) \in \mathcal{TC}(TS)$, we shall write $TC_1 \xrightarrow{TP} TC_2$, if $TC_1 \longrightarrow TC_2$ and TP is the isomorphism class of $(S_{TS}\lceil(C_2 \setminus C_1)\ , T_2|_{(C_2\setminus C_1)})$. The set $L_{tp}(TS) = \{TP \mid TC_{TS} \xrightarrow{TP} TC$ for some $TC \in \mathcal{TC}(TS)\}$ is the *tp-language* of TS. To illustrate the concept, we consider the *tp*-language of the timed event structure TS_1, shown in Fig. 1: $L_{tp}(TS_1) = \{(\mathcal{O}, 0),\ \overset{[d_1,d_1]}{a},\ \overset{[d_2,d_2]}{b},\ \begin{matrix}\overset{[d_3,d_3]}{a}\\ \underset{b}{[d_2,d_2]}\end{matrix},\ \overset{[d_4,d_4]}{a} \longrightarrow \overset{[d_5,d_5]}{b} \mid d_1, d_4 \in [3,5],\ d_2, d_5 \in [4,5],\ d_3 \in [3,6],\ d_4 \leq d_5\}$.

Using *tp*-languages and leading relations of the form $\xrightarrow{TP}$, we obtain timed pomset trace equivalence (denoted $\equiv_{tp}$) and timed pomset bisimulation equivalence (denoted $\leftrightarrow_{tp}$) exactly as the corresponding equivalences in Definition 4. Timed pomset bisimulation is clearly stronger than timed pomset trace equivalence.

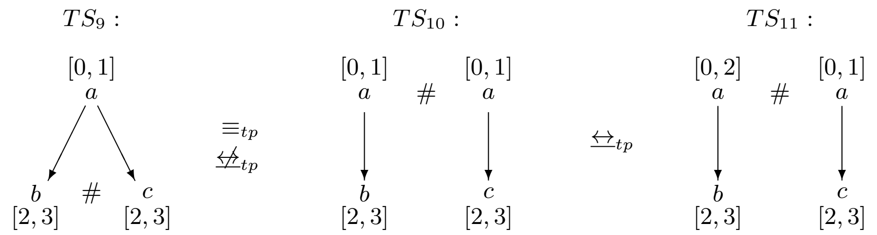


Fig. 4.

Considering the timed event structures, shown in Fig. 3 and 4, we obtain $TS_9 \equiv_{tp} TS_{10}$, while $TS_7 \not\equiv_{tp} TS_8$, since, for example, $\overset{[1,1]}{a} \longrightarrow \overset{[1,1]}{b} \in L_{tp}(TS_8)$ and $\overset{[1,1]}{a} \longrightarrow \overset{[1,1]}{b} \notin L_{tp}(TS_7)$. Further, we have $TS_{10} \underline{\leftrightarrow}_{tp} TS_{11}$, but $TS_9 \not\underline{\leftrightarrow}_{tp} TS_{10}$, because in the timed configuration of TS_9, containing only the timed action $(a, 1)$, the executions of the timed pomset $\overset{[2,2]}{b}$ and the timed pomset $\overset{[2,2]}{c}$ are possible, but it is not the case in TS_{10}.

6 Comparison of Equivalences

The common framework used to define different observational equivalences allows us to study the relationships between the three induced semantics. The theorems we state in the section are a step towards a better understanding of the interrelations between interleaving, step, and partial order semantics. In particular, we will give the hierarchy for the equivalences and will establish that some of them coincide on particular subclasses of timed event structures.

Theorem 1 *Let TS and TS' be timed event structures. Then*

(i) $TS \equiv_{ti} TS' \Leftarrow TS \equiv_{ts} TS' \Leftarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ti} TS' \Leftarrow TS \underline{\leftrightarrow}_{ts} TS' \Leftarrow TS \underline{\leftrightarrow}_{tp} TS'$.

Proof Sketch. Immediately follows from the definitions of the equivalences. □

The timed event structures shown in Fig. 2-4 show that the converse implications of the above theorem do not hold and that the six equivalences are all different.

Now one can ask the obvious question what happens with all these equivalences if we restrict ourselves to some subclasses of the model under consideration. A timed event structure $TS = (S = (E, \leq, \#, l), D)$ is called *sequential*, if $\smile_S = \emptyset$; TS is *deterministic*, if $e \smile_S e'$ or $e \#^1_S e' \Rightarrow l(e) \neq_S l(e')$ and $D_{TS}(e) \cap D_{TS}(e') \neq \emptyset$.

The next theorem shows that if we only consider timed event structures which represent timed sequential processes then all the three semantics of timed trace and timed bisimulation equivalences coincide.

Theorem 2 *Let TS and TS' be sequential timed event structures. Then*

(i) $TS \equiv_{ti} TS' \Rightarrow TS \equiv_{ts} TS' \Rightarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ti} TS' \Rightarrow TS \underline{\leftrightarrow}_{ts} TS' \Rightarrow TS \underline{\leftrightarrow}_{tp} TS'$.

Proof Sketch. We shall show that $TS \equiv_{ti} TS'$ implies $TS \equiv_{ts} TS'$ (the remaining cases are similar). Asssume $TS \equiv_{ti} TS'$. Take an arbitrary sequence $w = (A_1, d_1) \ldots (A_n, d_n) \in L_{ts}(TS)$. We have to show that $w \in L_{ts}(TS')$. Since TS is a sequential timed event structure, we have: $\forall 0 \leq i \leq n \diamond A_i = \{a_i\}$ for some $a_i \in Act$. According to our assumption, it holds $(a_1, d_1) \ldots (a_n, d_n) \in L_{ts}(TS')$. This implies $w \in L_{ts}(TS')$. An arbitrary choice of w guarantees $TS \equiv_{ts} TS'$. □

The theorem below establishes that if we only consider timed event structures which represent deterministic processes then timed step and timed partial order semantics coincide.

Theorem 3 *Let TS and TS' be deterministic timed event structures. Then*

(i) $TS \equiv_{ts} TS' \Rightarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ts} TS' \Rightarrow TS \underline{\leftrightarrow}_{tp} TS'$.

Proof Sketch. We shall prove that $TS \underline{\leftrightarrow}_{ts} TS'$ implies $TS \underline{\leftrightarrow}_{tp} TS'$ (the proof of item (i) is simpler). Let $\mathcal{B}$ be a timed step bisimulation between $TS = (S, D)$ and $TS' = (S', D')$. Take an arbitrary pair $(TC_1 = (C_1, T_1),\ TC_2 = (C_2, T_2)) \in \mathcal{B}$. W.l.o.g. suppose $TC_1 = \widetilde{TC}_0 \overset{(\{a_1\},d_1)}{\longrightarrow} \widetilde{TC}_1 \cdots \widetilde{TC}_{n-1} \overset{(\{a_n\},d_n)}{\longrightarrow} \widetilde{TC}_n = TC'_1 = (C'_1, T'_1)$ $(n \geq 0)$. By the definition of timed step bisimulaion, we have: $TC_2 = \widehat{TC}_0 \overset{(\{a_1\},d_1)}{\longrightarrow} \widehat{TC}_1 \cdots \widehat{TC}_{n-1} \overset{(\{a_n\},d_n)}{\longrightarrow} \widehat{TC}_n = TC'_2 = (C'_2, T'_2)$ and $(\widetilde{TC}_i, \widehat{TC}_i) \in \mathcal{B}$ for all $0 \leq i \leq n$. Assume $\widetilde{TC}_i = (\widetilde{C}_i, \widetilde{T}_i)$ and $\widehat{TC}_i = (\widehat{C}_i, \widehat{T}_i)$ for all $0 < i < n$. Let $\widetilde{C}_i \setminus \widetilde{C}_{i-1} = \{e_i\}$, and $\widehat{C}_i \setminus \widehat{C}_{i-1} = \{e'_i\}$ for all $0 < i \leq n$. We shall show by induction on n that $(S\lceil(C'_1 \setminus C_1), T'_1\mid_{(C'_1 \setminus C_1)}) \simeq (S'\lceil(C'_2 \setminus C_2), T'_2\mid_{(C'_2 \setminus C_2)})$. The case when $n = 0$ is trivial. Suppose $n > 0$. Two cases are admissible. If $n = 1$, the result follows from the definition of a deterministic timed event structure. Assume $n > 1$. It sufficies to show that $e_i <_S e_j \iff e'_i <_{S'} e'_j$ for all $1 \leq i, j \leq n$ such that $i < j$. Suppose a contrary, i.e. for some $1 \leq i, j \leq n$ such that $i < j$ it holds: $e_i <_S e_j$ and $e'_i \smile_{S'} e'_j$ (the converse case when $e_i \smile_S e_j$ and $e'_i <_{S'} e'_j$ is similar). W.l.o.g. assume $j = i + 1$.

We shall show that there exists $\widehat{TC}'_{i+1} \in \mathcal{TC}(TS')$ such that $\widehat{TC}_{i-1} \overset{(\{a_i,a_{i+1}\},d')}{\longrightarrow} \widehat{TC}'_{i+1}$ for some $d' \in \mathbf{R}_0^+$. Take $\widehat{C}'_{i+1} = \widehat{C}_{i+1}$. Further, take $\widehat{T}'_{i+1} : \widehat{C}'_{i+1} \longrightarrow \mathbf{R}_0^+$ such that $\widehat{T}'_{i+1}|_{\widehat{C}_{i-1}} = \widehat{T}_{i-1}$ and $\widehat{T}'_{i+1}(e'_i) = \widehat{T}'_{i+1}(e'_j) = d'$, where $d' \in D'(e'_i) \cap D'(e'_j)$ such that $d_i \leq d \leq d_j$ (the existence of d' is guaranteed by the definitions of a deterministic timed event structure, the set $Interv$ and the relation $\rightarrow$ on timed configurations). We have to show that $\widehat{TC}'_{i+1} = (\widehat{C}'_{i+1}, \widehat{T}'_{i+1}) \in \mathcal{TC}(TS')$. Since $\widehat{TC}_{i-1} \in \mathcal{TC}(TS')$ and $d' \in D'(e'_i) \cap D'(e'_j)$, the truth of item (i) of Definition 3 is obvious. Due to the facts that $\widehat{TC}_i \in \mathcal{TC}(TS')$ and $d_i \leq d'$, item (ii) of Definition 3 holds, by the definition of the relation $\rightarrow$ on timed configurations. Since $\widehat{TC}_{i+1} \in \mathcal{TC}(TS')$ and $d' \leq d_j$, it is straightforward to show the truth of item (iii) of Definition of 3. According to the construction of $\widehat{TC}'_{i+1}$, it holds: $\widehat{TC}_{i-1} \overset{(\{a_i,a_{i+1}\},d')}{\longrightarrow} \widehat{TC}'_{i+1}$.

Hence, we have $\widetilde{TC}_{i-1} \overset{(\{a_i,a_{i+1}\},d_i)}{\longrightarrow} \widetilde{TC}'_{i+1}$, by the definition of timed step bisimulation. Then there exists an event $f \in E_S$ such that $f \neq e_{i+1}$ and $l_S(f) = a_{i+1}$. Consider all the possible relations between e_{i+1} and f. If $e_{i+1} <_S f$ $(e_{i+1} >_S f)$, then $\widetilde{C}'_{i+1}$ $(\widetilde{C}_{i+1})$ is not a confuguration. If $e_{i+1} \smile_S f$ or $e_{i+1} \#_S f$, then we get a contradiction to the definition of a deterministic timed event structure.

An arbitrary choice of $(TC_1,\ TC_2)$ guarantees $TS \equiv_{ps} TS'$. $\square$

The two rightmost timed event structures in Fig. 2 show that even for deterministic timed event structures there is a difference between timed interleaving and timed partial order semantics.

7 Conclusion

In this paper, we have given a flexible abstract mechanism, based on observational equivalences which allows us to consider timed event structures as the basis of three different approaches (sequences of actions, multisets of actions, partial orders of actions) to the description of concurrent and real time systems. The results obtained show that these three semantics in general provide formal tools with an increasing power. Furthermore, when dealing with particular subclasses of the model there is no difference between a more concrete or a more abstract approach.

In a future work, we plan to extend the obtained results to other observational equivalences (e.g., equivalences taking into account internal actions) of timed systems. Some investigation on different timed concurrent semantics of testing equivalence which is located between trace and bisimulation equivalences in the linear time – branching time spectrum is now under way and we plan to report on this work elsewhere.

References

1. Aceto L.: History preserving, causal and mixed-ordering equivalence over stable event structures *Fundamenta Informaticae* **17(4)** (1992) 319–331.
2. R. Alur, D. Dill. The theory of timed automata. *Theoretical Computer Science* **126** (1994) 183–235.
3. R. Alur, T.A. Henzinger. Logics and models of real time: a survey. *Lecture Notes in Computer Science* **600** (1992) 74–106.
4. M.V. Andreeva, E.N. Bozhenkova, I.B. Virbitskaite. Analysis of timed concurrent models based on testing equivalence. *Fundamenta Informaticae* **43(1-4)** (2000) 1–20.
5. P. Buchholz, I. Tarasyuk. A class of stochastic Petri nets with step semantics and related equivalence notions. Technische Berichte TUD-FI00-12, Technische Universitaet Dresden (2000).
6. K. Čerāns. Decidability of bisimulation equivalences for parallel timer processes. *Lecture Notes in Computer Science* **663** (1993) 302–315.
7. R.J. van Glabbeek. The linear time – branching time spectrum II: the semantics of sequential systems with silent moves. Extended abstract. *Lecture Notes in Computer Science* **715** (1993) 66–81.
8. R.J. van Glabbeek, U. Goltz. Equivalence notions for concurrent systems and refinement of actions. *Lecture Notes in Computer Science* **379** (1989) 237–248.
9. U. Goltz, H. Wehrheim. Causal testing. *Lecture Notes in Computer Science* **1113** (1996) 394–406.
10. M. Hennessy, R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM* **32** (1985) 137–162.

11. Hoare C.A.R. Communicating sequential processes. Prentice-Hall, London (1985).
12. J.-P. Katoen, R. Langerak, D. Latella, E. Brinksma. On specifying real-time systems in a causality-based setting. *Lecture Notes in Computer Science* **1135** (1996) 385–404.
13. D. Murphy. Time and duration in noninterleaving concurrency. *Fundamenta Informaticae* **19** (1993) 403–416.
14. Park D. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science* **104** (1981) 167–183.
15. Pinchinat, S.: Ordinal processes in comparative concurrency semantics. *Lecture Notes in Computer Science* **626** (1992).
16. B. Steffen, C. Weise. Deciding testing equivalence for real-time processes with dense time. *Lecture Notes in Computer Science* **711** (1993) 703–713.
17. F.W. Vaandrager. Determinism $\rightarrow$ (event structure isomorphism $=$ step sequence equivalence). *Theoretical Computer Science* **79(2)** (1991) 275–294.
18. G. Winskel. An introduction to event structures. *Lecture Notes in Computer Science* **354** (1988) 364–397.
19. C. Weise, D. Lenzkes. Efficient scaling-invariant checking of timed bisimulation. *Lecture Notes in Computer Science* **1200** (1997) 176–188.

The Impact of Synchronisation on Secure Information Flow in Concurrent Programs

Andrei Sabelfeld

Department of Computer Science
Chalmers University of Technology and University of Göteborg
412 96 Göteborg, Sweden
fax: +46 31 16 56 55, andrei@cs.chalmers.se

Abstract. Synchronisation is fundamental to concurrent programs. This paper investigates the security of information flow in multi-threaded programs in the presence of synchronisation. We give a small-step operational semantics for a simple shared-memory multi-threaded language with synchronisation, and present a compositional timing-sensitive bisimulation-based confidentiality specification. We propose a type-based analysis improving on previous approaches to reject potentially insecure programs.

1 Introduction

1.1 Motivation

This paper focuses on the problem of program *confidentiality*, i.e., determining whether a given shared-memory multi-threaded program has secure information flow. The program runs on data partitioned on *high* (private) and *low* (public) security data, although a more general lattice of security levels can be considered. The program is not trusted, and the program's low output is publicly available as well as, possibly, timing information about the program's execution. Such a program might be received over the Internet and legitimately send its low output over the Internet such that the timing of the program's Internet accesses is observable.

1.2 Background

The problem of confidentiality for various programming languages has been investigated by many researchers including [8,9,7,4,14,6,15,22,11,20,1,21,12,2,17, 18,16]. The issue of secure information flow has become especially important with the growing popularity of mobile code and networked information systems. For distributed programming, the use of multi-threaded programming languages has become extremely popular [5]. However, in the setting of a shared-memory multi-threaded language, the majority of investigations in the area of secure information flow, e.g., [11,20,21,17] do not treat *synchronisation* in the language. Although the security logic of [4] does consider synchronisation primitives, there is neither a soundness proof nor a decision algorithm given for the logic. Because synchronisation is fundamental to concurrent programs, it is highly desirable to have a robust security specification and tools that aid in the design of secure programs with synchronisation.

To bridge the gap, this paper presents a compositional bisimulation-based confidentiality specification for multi-threaded programs with synchronisation and proposes a type-based analysis improving on previous approaches to reject potentially insecure programs.

1.3 Insecure Flows to Eliminate

Let us exemplify the types of insecure information flow that are in the focus of this paper. Suppose h is a high security variable and l is a low security one. There are several ways to leak information from h to the low-level observer (attacker). The simple program $l := h$ is an example of a *direct* flow. The program $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$ features an *indirect* flow through branching on a high condition.

The attacker may deduce secret information from the timing behaviour of the program. The program

$$\mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while}\ h < \mathit{MaxInteger}\ \mathsf{do}\ h := h + 1)\ \mathsf{else\ skip}$$

exemplifies a *timing* leak. Notice that the program does not operate on low variables. Nevertheless, assuming the program's execution time is visible for the low-observer, this is clearly a potential leak. Indeed, in case the initial value of h was 1 the program would take more time to execute than otherwise. The program

$$\mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while\ True\ do\ skip})\ \mathsf{else\ skip}$$

is a variation of the timing leak called a *termination* leak. Observing the termination of the program reveals that h was not 1.

Blocking a thread can change the observable behaviour of a computation, e.g., its termination behaviour. If blocking depends on high data, then the attacker might learn secrets through the observable behaviour. Concrete examples of *synchronisation* leaks are postponed until Section 4 where concrete synchronisation primitives will be available.

1.4 Overview

The rest of the paper is organised as follows. Section 2 introduces the syntax and semantics of a multi-threaded language. Section 3 motivates and specifies a bisimulation-based security definition. Section 4 gives a type system for detecting secure programs and shows its correctness with respect to the security definition. Section 5 gives an example of secure programming with synchronisation. Section 6 concludes by discussing related work.

2 A Multi-threaded Language with Synchronisation

To set the scene, let us begin by defining the syntax and semantics of a simple multi-threaded language with dynamic thread creation and synchronisation. Semaphores are widely used as a synchronisation primitive in shared-memory languages. As pointed out by Andrews ([5], p.viii): *"Semaphores were the first high-level concurrent programming mechanism and remain one of the most important."* Semaphores are general enough to implement both *mutual exclusion* and *condition synchronisation*. In this section we introduce semaphore primitives and give a small-step operational semantics for the multi-threaded language.

2.1 Semaphores

A *semaphore* [10] is a special variable (call it *sem*) that can only be manipulated by two commands: $\mathsf{wait}(sem)$ and $\mathsf{signal}(sem)$. The value of *sem* ranges over nonnegative integers. Initially, the value is 0 for every semaphore. The $\mathsf{wait}(sem)$ command blocks until *sem* is positive. Once *sem* is positive, it gets decremented by 1. The $\mathsf{signal}(sem)$ command increments *sem* by 1.

2.2 Semaphores by Busy Waiting

One approach to introducing semaphores in the language is to use macrodefinitions and define the synchronisation primitives by while-loop-based *busy waiting* (as in, e.g., [4]):

$$\begin{aligned} \mathsf{wait}(sem) &= (\mathsf{while}\ sem = 0\ \mathsf{do}\ \mathsf{skip}); sem := sem - 1 \\ \mathsf{signal}(sem) &= sem := sem + 1 \end{aligned}$$

Although these definitions are intuitive, there are two major problems with them. First, a waiting process *occupies the CPU with idle spinning.* Also, *delay and decrement must be a single atomic action.* Otherwise two $\mathsf{wait}(sem)$ threads might succeed when the initial value of *sem* is 1! Atomicity may be hard to implement in a distributed setting (not to mention that timing behaviour will spin out of control since one atomic action no longer takes one time unit). *Blocked waiting*, which commonly underlies semaphore implementations, does not have the disadvantages above. It is important that we adapt blocked waiting, since the dynamics of thread (un)blocking in a program's execution needs to be explicitly modelled due to its potential to affect the program's security.

2.3 Semaphores by Blocked Waiting

In order to define blocked waiting, we will use a special pool of waiting processes. The idea is that blocked processes should be sleeping (as opposed to spinning in busy waiting) until the respective signal is sent.

The syntax of the language is given in Figure 1. Let $C, D, E, \ldots$ range over commands *Com*, and let $\overrightarrow{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$.

Vectors $\overrightarrow{C}, \overrightarrow{D}, \overrightarrow{E}, \ldots$ range over $\overrightarrow{Com} = \cup_{n \in \mathbb{N}} Com^n$, the set of thread pools (or programs). A *state* $s \in \mathbf{St}$ is a finite mapping from variables (including special semaphore variables) to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables, respectively. Define *low-equivalence* by $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same.

The small-step semantics is given by transitions between *configurations*, i.e., between pairs each containing a program and a state. The deterministic part of the semantics is defined by the transition rules in Figure 2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoonup$ transitions are deterministic. There are two kinds of deterministic transitions: unlabelled and labelled. Labels are used in the semantics in order to propagate (un)blocking information. The general form of an unlabelled deterministic transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C' \overrightarrow{D}, s' |\!\rangle$. Here, one step of computation that starts with a command C in a state s gives a new main thread C', a (possibly empty) vector of spawned processes $\overrightarrow{D}$ and a new state s'. Command $\mathsf{fork}(C, \overrightarrow{D})$ dynamically creates a new vector of processes $\overrightarrow{D}$ which run in parallel with the main thread C. This has the effect of adding the vector of threads to the configuration.

Let *sem* be a variable from the special set *Sem* of semaphore variables. The general form of a labelled deterministic transition is either $\langle\!| C, s |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| \langle\rangle, s' |\!\rangle$ or $\langle\!| C, s |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| C', s' |\!\rangle$ where the label α ranges over the set of possible labels $\alpha \in \{\otimes sem, \odot sem \mid sem \in Sem\}$. These labels correspond to blocking and unblocking upon a respective semaphore. The $\mathsf{wait}(sem)$ command emits the "block" label $\otimes sem$ in case the value of *sem* is 0. Otherwise it decreases the value of *sem*. The $\mathsf{signal}(sem)$ emits the "unblock" label $\odot sem$. The labels are propagated through the sequential composition to the top level.

The concurrent part of the semantics is presented in Figure 3. The $\rightarrow$ transitions are nondeterministic. The nondeterminism is resolved by the scheduler in a particular implementation. The scheduler might be probabilistic and history-dependent, but, aiming at scheduler-independent security, we make no assumptions about peculiarities of a particular scheduler. The $\rightarrow$ transitions have the form $\langle\!| \overrightarrow{C}, w, s |\!\rangle \rightarrow \langle\!| \overrightarrow{C}', w', s' |\!\rangle$ where configurations are equipped with queues of waiting processes $w, w' : Sem \rightarrow \overrightarrow{Com}$. Whenever the top level receives a $\otimes sem$ signal, the blocked thread is put in the end of the FIFO queue associated with *sem*. If the top level receives an $\odot sem$ signal, the first thread in the FIFO gets awakened or, in case, the FIFO is empty, the value of *sem* gets incremented.

We can extract a simple model of the timing behaviour of multi-threaded programs from the small-step semantics. This is done by assuming that each $\rightharpoonup$ transition takes a single unit of time to execute. This approach gives only a rough approximation of the actual timing behaviour, but simple extensions are possible in order to make it sensitive to the timing behaviour of particular commands (cf. [2]).

$$\begin{aligned} C ::= \; & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C, \overrightarrow{D}) \mid \mathsf{wait}(Sem) \mid \mathsf{signal}(Sem) \end{aligned}$$

Fig. 1. Command syntax

$$\langle\!| \mathsf{skip}, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \langle\rangle, s |\!\rangle \qquad \frac{\langle\!| exp, s |\!\rangle \downarrow n}{\langle\!| x := exp, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \langle\rangle, [x := n]s |\!\rangle}$$

$$\frac{\langle\!| C_1, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \langle\rangle, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C_2, s' |\!\rangle} \qquad \frac{\langle\!| C_1, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C_1' \overrightarrow{D}, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| (C_1'; C_2) \overrightarrow{D}, s' |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C_1, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C_2, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C; \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \langle\rangle, s |\!\rangle}$$

$$\frac{\langle\!| sem, s |\!\rangle \downarrow n \quad n > 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \langle\rangle, [sem := sem - 1]s |\!\rangle} \qquad \frac{\langle\!| sem, s |\!\rangle \downarrow 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \overset{\otimes sem}{\mathrel{-\!\!\triangleright}} \langle\!| \langle\rangle, s |\!\rangle}$$

$$\langle\!| \mathsf{fork}(C, \overrightarrow{D}), s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| C \overrightarrow{D}, s |\!\rangle \qquad \frac{\langle\!| C_1, s |\!\rangle \overset{\alpha}{\mathrel{-\!\!\triangleright}} \langle\!| C_1', s' |\!\rangle \quad \alpha \in \{\otimes sem, \odot sem\}}{\langle\!| C_1; C_2, s |\!\rangle \overset{\alpha}{\mathrel{-\!\!\triangleright}} \langle\!| C_1'; C_2, s' |\!\rangle}$$

$$\langle\!| \mathsf{signal}(sem), s |\!\rangle \overset{\odot sem}{\mathrel{-\!\!\triangleright}} \langle\!| \langle\rangle, s |\!\rangle$$

Fig. 2. Small-step deterministic semantics of commands

$$\frac{\langle\!| C_i, s |\!\rangle \mathrel{-\!\!\triangleright} \langle\!| \overrightarrow{C}, s' |\!\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} \overrightarrow{C} C_{i+1} \dots C_n \rangle, w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\otimes sem}{\mathrel{-\!\!\triangleright}} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \overrightarrow{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C_{i+1} \dots C_n \rangle, [w_{sem} := \overrightarrow{D} C']w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\mathrel{-\!\!\triangleright}} \langle\!| C', s' |\!\rangle \qquad w_{sem} = C \overrightarrow{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n C \rangle, [w_{sem} := \overrightarrow{D}]w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\mathrel{-\!\!\triangleright}} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \langle\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n \rangle, w, [sem := sem + 1]s' |\!\rangle}$$

Fig. 3. Concurrent semantics of thread pools

3 Security Specification

What is a secure program in the language we have just defined? This section focuses on defining confidentiality and motivating the chosen definition.

3.1 Noninterference

The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, the absence of dependencies). If direct, indirect, and timing flows are considered then, intuitively, a program has the extensional *noninterference* property if *varying the high input will not change the possible low-level observations*, i.e., low inputs, low outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully pursued the extensional view, including [7,22,11,20,1,21,12,2,17,18,16] for the justification of security analyses and verification techniques for different languages. We follow the extensional approach and focus on the extensional security for our language.

The main idea behind the bisimulation-based approach promoted by [17] is to formalise the indistinguishability of the behaviours of two programs C and D for the attacker by $C \sim_L D$, where $\sim_L$ is a *low-bisimulation*. Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply:

$$C \text{ is secure iff } C \sim_L C$$

For our purposes we adapt a variation of the *strong low-bisimulation* [17].

Definition 1 *Define the* strong *low-bisimulation* $\cong_L$ *to be the union of all symmetric relations* R *on thread pools of equal size, such that if* $\langle C_1 \dots C_n \rangle \ R \ \langle D_1 \dots D_n \rangle$ *then*

$$\forall s_1 =_L s_2 \forall i \forall \alpha. \langle\!| C_i, s_1 |\!\rangle \overset{\alpha}{\twoheadrightarrow} \langle\!| \overrightarrow{C}', s_1' |\!\rangle \Longrightarrow$$
$$\exists \overrightarrow{D}', s_2'. \langle\!| D_i, s_2 |\!\rangle \overset{\alpha}{\twoheadrightarrow} \langle\!| \overrightarrow{D}', s_2' |\!\rangle, \overrightarrow{C}' \ R \ \overrightarrow{D}', s_1' =_L s_2'$$

where $\alpha \in \{\epsilon, \otimes sem, \odot sem\}$. The security specification then is:

Definition 2 $\overrightarrow{C}$ *is* secure $\Longleftrightarrow \overrightarrow{C} \cong_L \overrightarrow{C}$.

One can show that the relation $\cong_L$ is transitive, but not reflexive. E.g., $l := h \not\cong_L l := h$, which reflects the fact that the program behaves differently for the attacker, depending on the initial value of h.

The choice of this bisimulation allows for robust security. As argued in [17], *strong low-bisimulation captures timing flows.*[1] If two commands may have a different timing behaviour depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. Also, *strong low-bisimulation is scheduler-independent.* Thus, our notion of security is robust with respect to any choice of a particular scheduler.

Thanks to the clear separation between deterministic and nondeterministic semantics, the underlying theory of [17] is applicable to the new semantics, which, e.g, suggests a generalisation to probabilistic schedulers (although it is outside the scope of this paper). Despite the fact that these features impose restrictions on what can be considered low-bisimilar, the choice of *strong low-bisimulation is adequate* (not too restrictive) for the type-based analysis proposed in Section 4. We will prove that whenever a command is typable in our system, it is secure.

4 Type-Based Security Analysis

This section presents an automatic compositional analysis for certifying secure programs, extending and improving on previous approaches [4,21,2,17]. The section concludes by stating soundness results.

4.1 The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct, indirect and synchronisation insecure information flows then it might be accepted by the system and transformed into a program that is also free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C}' : \vec{S}l$, where $\vec{C}$ is a program, $\vec{C}'$ is the result of its transformation and $\vec{S}l$ is the type of $\vec{C}'$. The type $\vec{S}l$ is $\vec{C}'$'s *low slice*, i.e., essentially a copy of $\vec{C}'$ in which assignments to (and conditionals on) high variables have been replaced by skip's. The low slice $\vec{S}l$ has no occurrences of h and models the timing behaviour of $\vec{C}'$, as observable by other threads.

The typing and transformation rules are presented in Figure 4. The variables h and l have the types *high* and *low* respectively. Value literals n may be considered as either *high* or *low*. An arbitrary expression *Exp* is considered *high*. In all but the If$_{high}$ rule, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. The command skip is its own low slice and therefore its own type. The rule Set$_{low}$ prevents direct insecure information flows—the assignment $l := h$ is not typable. The rule Set$_{high}$ types an assignment to the high variable with the low slice skip. The rules Exp$_{low}$, Seq, If$_{low}$, Par and Fork propagate types compositionally. The guard of the while-loop

[1] Nevertheless timing attacks at the layer below our assumptions (using, e.g., non-atomicity of semantics steps (cf. Section 2.3) or cache behaviour [3]) are still possible.

[Exp] $h : high \qquad l : low \qquad n : \tau \qquad Exp : high$

[Exp$_{low}$] $$\frac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$$

[Skip] $\mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$

[Set$_{low}$] $$\frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$$

[Set$_{high}$] $h := Exp \hookrightarrow h := Exp : \mathsf{skip}$

[Seq] $$\frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$$

[While] $$\frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

[Par] $$\frac{C_1 \hookrightarrow C_1' : Sl_1 \ \ldots\ C_n \hookrightarrow C_n' : Sl_n}{\langle C_1 \ldots C_n \rangle \hookrightarrow \langle C_1' \ldots C_n' \rangle : \langle Sl_1 \ldots Sl_n \rangle}$$

[Fork] $$\frac{C_1 \hookrightarrow C_1' : Sl_1 \quad \overrightarrow{C}_2 \hookrightarrow \overrightarrow{C}'_2 : \overrightarrow{S}l_2}{\mathsf{fork}(C_1, \overrightarrow{C}_2) \hookrightarrow \mathsf{fork}(C_1', \overrightarrow{C}'_2) : \mathsf{fork}(Sl_1, \overrightarrow{S}l_2)}$$

[Wait] $$\frac{sem : low}{\mathsf{wait}(sem) \hookrightarrow \mathsf{wait}(sem) : \mathsf{wait}(sem)}$$

[Signal] $$\frac{sem : low}{\mathsf{signal}(sem) \hookrightarrow \mathsf{signal}(sem) : \mathsf{signal}(sem)}$$

[If$_{low}$] $$\frac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$$

[If$_{high}$] $$\frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

Fig. 4. Security type system

in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard. The rules Wait and Signal guarantee that only low semaphores are used for synchronisation.

In addition to the insecure flows exemplified in Section 1, there are further ways to leak information through blocking. Synchronising on a high semaphore variable leads to (un)blocking of a thread and, clearly, may affect the termination behaviour of the program. As a consequence, neither $\mathsf{wait}(sem)$ nor $\mathsf{signal}(sem)$ on a high semaphore is allowed by the type system. The rule If$_{high}$ prevents indirect insecure flows and timing flows. An indirect flow might be performed through synchronisation on a low semaphore depending on a high guard, e.g., $\mathsf{if}\ h = 1\ \mathsf{then}\ \mathsf{wait}(sem)\ \mathsf{else}\ \mathsf{skip}$. Thus, the type system prevents from synchronisation in the branches of an if-on-high. Let $al(C)$ be a boolean function returning True whenever there is a syntactic occurrence of either an assignment to low variable l or a synchronisation primitive (wait or signal) in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) = \mathsf{False}$ prevents indirect leaks. Both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure (preventing timing leaks) it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other (cf. Figure 5). The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

4.2 A Modification of the Cross-Copying Rule

A modification of the rule If$_{high}$ might be used to guarantee that dummy computation is inserted "evenly" and that it does not block useful computation within the branches of the resulting program. Such a rule makes use of the parallel composition instead of the sequential one when cross-copying (cf. Figure 6). Such an alternative rule If$'_{high}$ is:

$$[\mathrm{If}'_{high}] \quad \frac{B : high \quad C_1 \hookrightarrow C'_1 : Sl_1 \quad C_2 \hookrightarrow C'_2 : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\begin{array}{r}\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \mathsf{fork}(C'_1; \mathsf{wait}(sem),\ Sl_2; \mathsf{signal}(sem)) \\ \mathsf{else}\ \mathsf{fork}(Sl_1; \mathsf{wait}(sem),\ C'_2; \mathsf{signal}(sem)) \\ : \mathsf{skip}; \mathsf{fork}(Sl_1; \mathsf{wait}(sem),\ Sl_2; \mathsf{signal}(sem))\end{array}}$$

where sem is a fresh (low) semaphore, unused elsewhere in the program. The use of the semaphore in the rule ensures that both main threads in each branch of the resulting code have terminated before passing the control outside the if. This is necessary to prevent potential changes in the order of execution of the if and the following commands.

Interestingly, the alternative rule turns out to be less restrictive, as demonstrated in the following example. Let us denote loop = while True do skip. Consider the program on the left in Figure 7. Transforming the program by the rule If$_{high}$ (as in the type systems of [2,17]) gives a program that gets stuck with the dummy while-loop (which appears in the first position in the sequential composition in both branches). This implies that the fragment $h := -h$ never gets to be executed. The result of the transformation is on the right in Figure 7. (In the examples we omit the low slices for simplicity.) As depicted in Figure 8, the modified rule If$'_{high}$ transforms the program into a program where the high computation (the fragment $h := -h$) is enabled while the dummy loop executes in parallel.

4.3 Correctness of the Analysis

The key to straightforward correctness proofs is the *compositionality of the security property* (Definition 2). In the standard security terminology, this is called the *hook-up* property [13]. Having the hook-up property facilitates modular development of secure code.

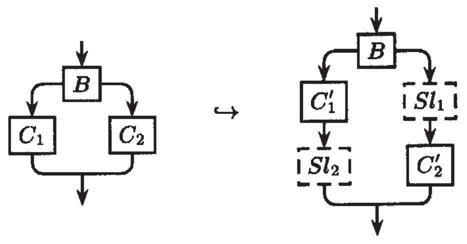


Fig. 5. Padding in rule If$_{high}$

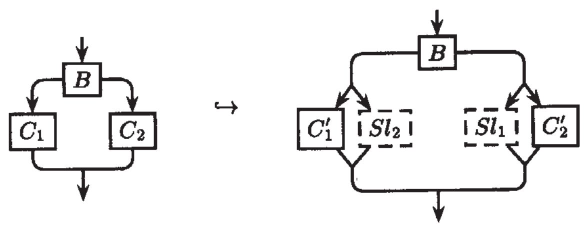


Fig. 6. Padding in rule If$'_{high}$

Termination leak		**Padded transformation result**
if $h > 0$ then loop else $h := -h$	$\hookrightarrow$	if $h > 0$ then loop; skip else loop; $h := -h$

Fig. 7. Example of applying If$_{high}$

Termination leak		**Padded transformation result**
if $h > 0$ then loop else $h := -h$	$\hookrightarrow$	if $h > 0$ then fork(loop; wait(sem), skip; signal(sem)) else fork(loop; wait(sem), $h := -h$; signal(sem))

Fig. 8. Example of applying If$'_{high}$

Clearly, an arbitrary context is not necessarily secure, i.e., the low-bisimulation is not a congruence. The program $l := h$ is an example of an insecure ground context (recall that $l := h \not\approxeq_L l := h$). Let us go through the program constructs of the language and investigate what constraints they impose for building *secure contexts*. We identify secure contexts and give a formal proof.

Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet}_1, \vec{\bullet}_2]$ is secure iff it has one of these forms:

$$\begin{aligned}\mathbb{C}[\vec{\bullet}_1, \vec{\bullet}_2] ::= {} & \mathsf{skip} \mid h := \mathit{Exp} \mid l := \mathit{Exp}\ (\mathit{Exp} \text{ is low}) \\ & \mid [\bullet_1]; [\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low}) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low}) \mid \mathsf{fork}([\bullet_1], [\vec{\bullet}_2]) \mid \langle[\vec{\bullet}_1][\vec{\bullet}_2]\rangle \\ & \mid \mathsf{wait}(\mathit{sem})\ (\mathit{sem} \text{ is low}) \mid \mathsf{signal}(\mathit{sem})\ (\mathit{sem} \text{ is low})\end{aligned}$$

where a (boolean or arithmetic) expression Exp is defined to be *low* iff $\forall s_1 =_L s_2.\, \exists n.\, \langle \mathit{Exp}, s_1\rangle \downarrow n \;\&\; \langle \mathit{Exp}, s_2\rangle \downarrow n$. Otherwise, the expression is *high*.

Theorem 1 (Secure congruence) *Let $\vec{C}_1 \approxeq_L \vec{C}'_1$ and $\vec{C}_2 \approxeq_L \vec{C}'_2$. If $\mathbb{C}[\vec{\bullet}_1, \vec{\bullet}_2]$ is a secure context, then $\mathbb{C}[\vec{C}_1, \vec{C}_2] \approxeq_L \mathbb{C}[\vec{C}'_1, \vec{C}'_2]$. If $\mathbb{C}[\bullet_1, \bullet_2]$ is an if-on-high context $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2] \approxeq_L \mathbb{C}[C'_1, C'_2]$ provided $C_1 \approxeq_L C_2$.*

Due to the separation of the semantics into deterministic and nondeterministic parts, we are able to reuse the proof of secure congruence for the language without synchronisation from [17] since it only operates on the deterministic semantics. Adding the secure contexts $\mathsf{wait}(\mathit{sem})$ and $\mathsf{signal}(\mathit{sem})$ (for a low sem) poses no substantial problems. An immediate corollary is the hook-up result.

Corollary 1 (Hook-up) *If $\vec{C}_1$ and $\vec{C}_2$ are secure and $\mathbb{C}[\vec{\bullet}_1, \vec{\bullet}_2]$ is a secure context, then $\mathbb{C}[\vec{C}_1, \vec{C}_2]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided $C_1 \approxeq_L C_2$.*

Proof. Assuming $\vec{C}_1$ and $\vec{C}_2$ are secure, we have $\vec{C}_i \approxeq_L \vec{C}_i$ for $i = 1, 2$. By the security congruence theorem $\mathbb{C}[\vec{C}_1, \vec{C}_2] \approxeq_L \mathbb{C}[\vec{C}_1, \vec{C}_2]$, i.e., $\mathbb{C}[\vec{C}_1, \vec{C}_2]$ is secure. □

Since both the security property and the analysis are compositional, the correctness proof is a simple structural induction. We have the following theorem.

Theorem 2 $\vec{C} \hookrightarrow \vec{C}' : \vec{S}l \Longrightarrow \vec{C}' \approxeq_L \vec{S}l.$

Corollary 2 (Correctness of the Analysis) $\vec{C} \hookrightarrow \vec{C}' : \vec{S}l \Longrightarrow \vec{C}'$ *is secure.*

To see why the corollary holds, note that Theorem 2 gives $\vec{C}' \approxeq_L \vec{S}l$. The symmetry and transitivity of $\approxeq_L$ entails $\vec{C}' \approxeq_L \vec{C}'$, i.e., $\vec{C}'$ is secure.

4.4 Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? Clearly, the padding introduced by the transformation can change the timing, but otherwise it is just additional "stuttering". To make this point precise, let us define a *weak (bi)simulation* on configurations. Let $\langle\!|C, s|\!\rangle \rightarrow^{\leq 1} \langle\!|C', s'|\!\rangle$ hold iff $\langle\!|C, s|\!\rangle = \langle\!|C', s'|\!\rangle$ or $\langle\!|C, s|\!\rangle \rightarrow \langle\!|C', s'|\!\rangle$.

Definition 3 *Define the weak simulation $\preceq$ (resp., bisimulation $\simeq$) to be the union of all (resp., symmetric) relations R on thread pools such that if $\overrightarrow{C}\ R\ \overrightarrow{D}$ and $\forall sem.\, w_{sem}\ R\ v_{sem}$ then for all s, s', w', C' there exist D' and v' such that*

$$\langle\!|\overrightarrow{C}, w, s|\!\rangle \rightarrow \langle\!|\overrightarrow{C}', w', s'|\!\rangle \Longrightarrow \langle\!|\overrightarrow{D}, v, s|\!\rangle \rightarrow^{\leq 1} \langle\!|\overrightarrow{D}', v', s'|\!\rangle, \\ \overrightarrow{C}'\ R\ \overrightarrow{D}', \forall sem.\, w'_{sem}\ R\ v'_{sem}$$

Lemma 1 *$\preceq$ and $\simeq$ are reflexive, transitive and respected by all contexts (not necessarily secure), i.e., $\preceq$ is a precongruence and $\simeq$ is a congruence.*

Theorem 3 $\overrightarrow{C} \hookrightarrow \overrightarrow{C}' : \overrightarrow{S}l \Longrightarrow \overrightarrow{C}' \preceq \overrightarrow{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Set$_{low}$, Set$_{high}$, Wait and Signal follow from the reflexivity of $\preceq$. The proof for Seq, While, Par, Fork and If$_{low}$ is by the induction hypotheses and the congruence of $\preceq$.

The remaining case is If$_{high}$ (reasoning for If$'_{high}$ is analogous). We have $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C'_1; Sl_2\ \mathsf{else}\ Sl_1; C'_2 : \mathsf{skip}; Sl_1; Sl_2$. By induction, $C'_1 \preceq C_1$ and $C'_2 \preceq C_2$. The side condition guarantees that there are no assignments or synchronisation in the slices Sl_1 and Sl_2 of the branches C_1 and C_2. Thus, Sl_1 and Sl_2 contain only (potentially nonterminating) dummy computation without changing the state. The insertion of such a computation might (infinitely) slow down the computation. By the definition of $\preceq$, we have $C'_1; Sl_2 \preceq C_1$ and $Sl_1; C'_2 \preceq C_2$. The congruence of $\preceq$ yields $C' \preceq C$. □

5 Secure Programming with Synchronisation

Despite restrictions imposed by the security requirements, one can still write useful programs with synchronisation. This section gives a simple example of a secure program that uses synchronisation.

Let us use *split binary semaphores* [5] in an example implementation of simple web-form processing. Suppose web forms are used for registering users of a security-sensitive website. As an entry gets processed, the low and high outputs are computed. In particular, the low variable *counter* is incremented. The high variable *milcounter* is incremented whenever the affiliation of the user is military. The program fragment is given in Figure 9.

The high variables are *record*, *buf*, *result* and *milcounter*. The low variables are *counter* and the semaphore variables *empty* and *full*. We assume a language

```
Main : $C$; fork($D_1, D_2$)
   $C$ : $empty := 1;\ full := 0;\ counter := 0;\ milcounter := 0$
  $D_1$ : while True
          do ... /* produce a new record when a new form is submitted */
             wait($empty$);
             $buf := record$; /* the record is now in the buffer */
             signal($full$)
  $D_2$ : while True
          do wait($full$);
             $result := buf$; /* fetch the record */
             signal($empty$);
             if $result.org$ = military
                then $milcounter := milcounter + 1$
                 else skip
             $counter := counter + 1$;
```

Fig. 9. Example of secure programming with synchronisation

Ref	**NI**	**Dec**	**Sync**	**Time**
Andrews & Reitman [4]	No	No	Yes	No
Heintze & Riecke [11]	No	Yes	No	No
Volpano & Smith [21]	Yes	Yes	No	protect
Agat [2]	Yes	Yes	No	Yes
Sabelfeld & Sands [17]	Yes	Yes	No	Yes
This paper	Yes	Yes	Yes	Yes

Fig. 10. Approaches to security analysis of multi-threaded programs

with enriched data structures (records) for the purpose of this example, but this does not affect the security argument. The program is accepted by the type system. Thus, the program is secure and, in particular, free of timing leaks.

6 Conclusions and Related Work

We have investigated the security of information flow in multi-threaded programs in the presence of synchronisation. The main result is that allowing neither synchronisation on high nor any synchronisation (not even on low) in the branches of an if-on-high is sufficient for building up a compositional timing-sensitive security specification from previous definitions that did not consider synchronisation. We have also proposed a type-based analysis that certifies programs according to the security specification. Let us conclude by discussing some improvements of this analysis compared to previous certification systems for security. Figure 10 gives a comparative overview of some of the most related approaches to analysing confidentiality of multi-threaded programs. The first column **Ref** gives references to the related systems. **NI** means whether the system has been proved to certify secure programs under an extensional noninterference-like security property. The third and fifth systems have been proved to guarantee *probabilistic noninterfer-*

ence. We expect probabilistic noninterference to hold for the presented system (in fact, it has been designed with probabilistic noninterference in mind). Future work includes developing a soundness proof with respect to probabilistic noninterference.

Dec stands for whether the system is given with a decision algorithm. All but the first system are decidable type systems whereas the first system is formulated as a logic with no decision algorithms or soundness proofs. **Sync** indicates whether the underlying language has synchronisation primitives. Only the first and the present investigations consider synchronisation. Furthermore, Agat [2] only considers sequential programs.

Time says whether the system captures timing covert channels. Andrews and Reitman [4] sketch the possibility of taking into account timing leaks in their security logic. However, the proposed mechanism rejects all programs that branch on high variables. Heintze and Riecke [11] stress the importance of timing-style attacks in a concurrent setting but do not give a proof that the type system accepts programs free of such attacks. Volpano and Smith consider timing flows in [21]. They allow branching on high by requiring all if-on-high commands to be embraced by special protect commands. The protect executes atomically by definition, making the timing difference invisible for the attacker. Such a command seems difficult to implement without locking the execution of every atomic command in the language or, as suggested by Smith [19], using additional mechanisms like thread priorities. Even that will not close external timing leaks, i.e., a time-consuming computation will not be hidden by a protect from the attacker with a stop-watch.

Acknowledgements. Thanks are due to David Sands for fruitful discussions and to Johan Agat and Makoto Takeyama for many valuable comments.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.
2. J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.
3. J. Agat and D. Sands. On Confidentiality and Algorithms. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, May 2001.
4. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.
5. Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.
6. J.-P. Banatre, C. Bryce, and D. Le Metayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.
7. E. S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
8. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
10. E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.
11. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.
12. K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1), 2000.
13. D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
14. M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.
15. P. Ørbæk. Can you Trust your Data? In *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, May 1995.
16. A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs.* PhD thesis, Chalmers University of Technology and Göteborg University, May 2001.
17. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000.
18. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.
19. G. Smith. Personal communication, 2000.
20. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 355–364, 19–21 January 1998.
21. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
22. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Dynamical Priorities without Time Measurement and Modification of the TCP

Valery A. Sokolov and Eugeny A. Timofeev

Yaroslavl State University,
150 000,Yaroslavl, Russia
{sokolov, tim}@uniyar.ac.ru

Abstract. A priority discipline without time measurements is described. The universality of this discipline is proved. The application of this discipline for the performance of TCP is shown.

1 Introduction

In this paper we describe a dynamic (changing in time) priority discipline of a service system, which does not require time measurements. We prove that this service discipline has all possible queue mean lengths for a one-device service system with ordinary flows of primary customers and branching flows of secondary customers.

The use of the discipline for organizing the work of the — (Transmission Control Protocol) [1] allows us to eliminate some TCP disadvantages, such as: the interpretation of a lost packet as the network overload, bulk transfer and so on. In this case, there is no need to make any changes in the TCP.

There are some works (see, for example, [3], [2]) which improve the TCP. The works [4], [5] propose a modification " with an adaptive rate" (ARTCP), which improves the TCP, but requires the introduction of an additional field to the protocol, namely, the round trip time interval.

2 Theoretical Results

In this section we describe a service discipline (the rule of selecting packets) without time measurements. We call it a probabilistic-divided service discipline. For the one-device service system with ordinary flows of primary customers and branching flows of secondary customers we show that this discipline has all possible queue mean lengths.

2.1 Probabilistic-Divided Service Discipline

Now we introduce a probabilistic-divided service discipline [8].

Let our system have n types of customers. The parameters of this discipline are

– a cortege of $N = 2n$ natural numbers $\mathbf{a} = (a_1, a_2, \ldots, a_N)$ such that

$$\forall i \in \{1, 2, \ldots, n\} \quad \exists 1 \leq f(i) < l(i) \leq N : a_{f(i)} = a_{l(i)} = i; \tag{1}$$

– N real numbers $\mathbf{b} = (b_1, \ldots, b_N)$ $(0 \leq b_i)$, such that

$$b_{f(i)} + b_{l(i)} = 1 \quad \forall i \in \{1, 2, \ldots, n\}. \tag{2}$$

Define the priority p_i $(1 \leq p_i \leq N)$ of every new customer of the type i $(1 \leq i \leq n)$ (arrival or appear after branching) as

$$p_i = \begin{cases} f(i), \text{ with the probability } & b_{f(i)}; \\ l(i), \text{ with the probability } & b_{l(i)}; \end{cases} \tag{3}$$

where $f(i), l(i)$ are defined in 1.

For the service we select the customer with the least priority.

2.2 Model

The service system has proven to be a useful tool for system performance analysis. In this section we have extended the application of such a system by incorporating populations of branching customers: whenever a customer completes the service, it is replaced by ν customers, where ν has a given branching distribution [7].

This single-server system is defined as follows:

– customers arrive according to a Poisson process with a rate λ;
– a newly arriving customer receives a type i with probability β_i, $(\beta_i \geq 0, \beta_1 + \ldots + \beta_n = 1)$;
– the service is nonpreemptive;
– the durations of service are independent random variables with the distribution function $B_i(t)(B_i(0) = 0)$ for i-type customers; denote by

$$b_i = \int_0^\infty t dB_i(t) < \infty, \quad b_{i2} = \int_0^\infty t^2 dB_i(t) < \infty; \tag{4}$$

– whenever the i-type customer completes service, it is replaced by ν_i customers, where $\nu_i = (k_1, k_2, \ldots, k_n)$ with probability $q_i(k_1, k_2, \ldots, k_n)$; by $Q_i(\mathbf{z}) = Q_i(z_1, z_2, \ldots z_n)$ denote a generating function of ν_i and denote by

$$q_{ij} = \frac{\partial Q_i}{\partial z_j}(1, \ldots, 1) < \infty, \qquad \frac{\partial^2 Q_i}{\partial z_j \partial z_k}(1, \ldots, 1) < \infty, \tag{5}$$

(because of the applications being modelled, our interest is confined to a system in which a branching process ν_i generates a total population with a finite expected size);
– new customers are immediately feed back to the ends of the corresponding queues;

– if the queue is not empty, the server immediately selects (by the service discipline) a customer and running;
– by γ_i we denote the expected total amount of service provided to a customer of type i and all its population; denote by

$$\rho = \lambda \sum_{i \in \Omega} \beta_i \gamma_i < 1. \tag{6}$$

Let

$$L_i = \lim_{T \to \infty} \frac{1}{T} \int_0^\infty \mathbf{E} l_i(t) dt,$$

where $l_i(t)$ is the amount of i-type customers in the queue at a moment t ($1 \leq i \leq n$).

Let $L_i(\mathbf{a}, \mathbf{b})$ be the mean length of customers of type i with the probabilistic-divided service discipline.

Our main theorem generalizes and improves the result of [8].

Theorem 1. *Let $L_1^*, \ldots, L_n^*$ be a mean length of queues under the stable regime with any service discipline, then there exist parameters $\mathbf{a}$ and $\mathbf{b}$ of the probabilistic-divided service discipline, such that $L_i(\mathbf{a}, \mathbf{b}) = L_i^*$, $i = 1, 2, \ldots, n$.*

3 Proof of the Theorem

To prove this theorem, we need some notation.

Let λ_i be the rate of type i customers (arrive or appear after branching). Then

$$\lambda_i = \sum_{j=1}^{n} q_{ji} \lambda_j + \lambda \beta_i, \quad i = 1, 2, \ldots, n. \tag{7}$$

The proof is found in [7].

In paper [9], the second author has proved that

$$\begin{aligned} &\textstyle\sum_{i=1}^{n} L_i^* \gamma_i = \omega(\{1, 2, \ldots, n\}), \\ &\textstyle\sum_{i \in S} L_i^* \gamma_i(S) \geq \omega(S), \quad \forall S \subset \{1, 2, \ldots, n\}, \end{aligned} \tag{8}$$

where $\gamma_i(S)$ is the expected total amount of service provided to a customer of type $i \in S$ and all its population in S ($\gamma_i(\{1, 2, \ldots, n\}) = \gamma_i$), and

$$\omega(S) = \inf_{a,b} \sum_{i \in S} L_i(a, b) \gamma_i(S).$$

Without loss of generality it can be assumed that

$$L_1^*/\lambda_1 \leq L_2^*/\lambda_2 \leq \ldots \leq L_n^*/\lambda_n. \tag{9}$$

Under the conditions of the theorem and (9), we describe an algorithm for finding parameters $\mathbf{a}$ and $\mathbf{b}$ of the probabilistic-divided service discipline.

STEP 0. Let

$$\begin{aligned} a_1 &= 1; \\ a_{2i} &= i+1,\ 1 \le i \le n-1; \\ a_{2i+1} &= i, \quad\ \ 1 \le i \le n-1; \\ a_{2n} &= n; \end{aligned}$$

$$\mathbf{b_0} = (0,0,1,0,1,0,1,0,1,0,\dots,0,1,1),$$

and

$$L_i^0 = L_i(\mathbf{a}, \mathbf{b_0}), \quad i = 1,2,\dots,n.$$

By S denote a subset of $\{1,2,\dots,n\}$ such that

$$S = \{i : L_i^0 \ge L_i^*\}.$$

STEP 1. Solving the system of equations

$$L_i(\mathbf{a},\mathbf{b}) = (1-t)L_i^0 + tL_i^*, i = 1,2,\dots,n, \tag{10}$$

with the complement condition

$$\text{if } j \in S \text{ then } b_j = 1 \text{ else } b_j = 0,$$

we get the solution $(t_j, \mathbf{d})$ for $j = 2,3,\dots,n$.

Let

$$t_\alpha = \min_{2 \le j \le n} \{t_j\}.$$

If $t_\alpha = 1$, we find parameters $\mathbf{a}$ and $\mathbf{b}$ of the probabilistic-divided service discipline.

STEP 2. Let $\alpha \in S$ (therefore $d_\alpha = 0$).

Let

$$m = \min\{i : i > l(\alpha), a_i \in S, a_i > \alpha, l(a_i) = i\},$$

$$k = \max\{i : i > m, a_i \notin S, a_i < \alpha\}.$$

If the maximum is not defined, we set $k = m$.

Displacing $a_{f(\alpha)} = \alpha$ after a_k, we get a new cortege

$$\mathbf{a} = (a_1, \dots, a_{f(\alpha)-1}, a_{f(\alpha)+1}, \dots, a_k, \alpha, a_{k+1}, \dots, a_{2n}).$$

Go to STEP 1.

STEP 3. Let $\alpha \notin S$ (therefore $d_\alpha = 1$).

Let

$$m = \max\{i : i < f(\alpha), a_i \in S, a_i < \alpha, f(a_i) = i\},$$

$$k = \min\{i : i < m, a_i \notin S, a_i > \alpha\}.$$

If the minimum is not defined, we set $k = m$.

Displacing $a_{l(\alpha)} = \alpha$ before a_k, we get a new cortege

$$\mathbf{a} = (a_1, \dots, a_{k-1}, \alpha, a_k, \dots, a_{l(\alpha)-1}, a_{l(\alpha)+1}, \dots, a_{2n}).$$

Go to STEP 1.

The proof that such a new cortege always exists is found in [9].

4 Modification of the TCP

The main idea of the suggested TCP modification is the following: in the ARTCP (considered in [4], [5]) the round trip time interval is replaced by the length of the corresponding queue.

The experimental testing of the ARTCP, carried out in the paper [6], has shown its efficiency. At the same time we can apply theorem 1, which shows that on average our suggested modification of the TCP has the same possibilities as the ARTCP. Thus, our modification improves the TCP, but does not require the introduction of an additional field in the protocol.

Let us consider our TCP modification. Without entering in details of the ARTCP (see [4], [5]) we shall describe its main characteristic feature, namely, the packet transmission rate. It is exactly the feature we will modify. Other components of the ARTCP will be without changing.

Let us define the rate that is used in the ARTCP (see [4], [5]). For every sent packet (in the Round Trip Time (RTT) interval) the ARTCP memorizes the value of $\tau = t_1 - t_0$, where t_0 is equal to the time of the current packet sending off and t_1 is equal to the time of the next packet sending off. The receiver memorizes the value of τ in the corresponding field. Then the rate has the following value

$$R = S/\tau,$$

where S is the length of the corresponding packet.

In the suggested modification the value τ is replaced by the value l_i/λ_i, where l_i is the length of the queue from the packet set of the given type i and λ_i is defined in (7).

Instead of rate changing we change the corresponding parameter b_i. The increment of the parameter b_i leads to the increment of the rate R.

If $b_i = 0$, then we change the parameters **a**, as described in STEP 2. The parameter **b** will be set up in its initial value.

If $b_i = 1$, then we change the parameters **a**, as described in STEP 3. The parameter **b** will be set up in its initial value.

5 Conclusion

Thus, the described construction allows us to eliminate the measurement in the service system. The application of this result to the TCP simplifies the ARTCP and doesn't require an additional field in the protocol. As a result, the use of probabilistic-divided discipline for organizing the work of the — eliminates some TCP disadvantages.

References

1. Stevens W.R. : TCP/IP Illustrated. Volume 1: The Protocols. Addison-Wesley, New York (1994)

2. Balakrishnan H., Padmanabhan V., Katz R. : The Effects of Asymmetry on TCP Performance. ACM MobiCom, **9** (1997)
3. Brakmo L., O'Malley S., Peterson L. : TCP Vegas: New Techniques for Congestion Detection and Avoidance. ACM SIGCOMM **8** (1994) 24-35
4. Alekseev I.V., Sokolov V.A. : The Adaptive Rate TCP. Model. and Anal. Inform. Syst. **6**, N 1 (1999) 4–11
5. Alekseev I.V., Sokolov V.A. : Compensation Mechanism for Adaptive Rate TCP. First IEEE/Popov workshop on Internet Technologies and Services, (October 25-28, 1999), Proceedings. **2** (1999) 68-75
6. Alekseev I.V. : Model and Analysis Transmission Protocol ARTCP. Investigation in Russia **27** (2000) 395–404 http://zhurnal.ape.relarn.ru/articles/2000/027.pdf
7. Kitaev M.Yu., Rykov V.V. : A Service System with a Branching Flow of Secondary Customers. Avtomatika i Telemechanika **9** (1980) 52–61
8. Timofeev E.A. : Probabilistic-divided service discipline and the polyphedron of mean waits in the system $GI|G|1$. Avtomatika i Telemechanika **10** (1991) 121–125
9. Timofeev E.A. : Optimization of the mean lengths of queues in the service system with branching secondary flows of customers. Avtomatika i Telemechanika **3** (1995) 60–67

From ADT to UML-Like Modelling
Short Abstract

Egidio Astesiano, Maura Cerioli, and Gianna Reggio

DISI-Universita' di Genova
Via Dodecaneso 35
16146 Genova, Italy
fax: +39-010-3536699
astes@disi.unige.it

UML and similar modelling techniques are currently taking momentum as a de facto standard in the industrial practice of software development. As other Object Oriented modelling techniques, they have benefited from concepts introduced or explored in the field of Algebraic Development Techniques — for short ADT, formerly intended as Abstract Data Types — still an active area of research, as demonstrated by the CoFI project. However, undeniably, UML and ADT look dramatically different, even perhaps with a different rationale. We try to address a basic question: can we pick up and amalgamate the best of both? The answer turns out to be not straightforward. We analyze correlations,lessons and problems. Finally we provide suggestions for further mutual influence, at least in some directions.

Transformation of UML Specification to XTG

Ella E. Roubtsova[1], Jan van Katwijk[2], Ruud C.M. de Rooij[2], and Hans Toetenel[2]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands
E.Roubtsova@tue.nl
[2] Faculty of Information Technology and Systems, Delft University of Technology
The Netherlands

Abstract. We use tuples of extended class, object and statechart UML-diagrams as UML specifications of real-time systems. The semantics of the UML specification is defined by transformation to the eXtended Timed Graphs (XTG). The correctness of our transformation is demonstrated by showing that the XTG computation tree can be projected into the computation tree of the corresponding UML specification. The transformation opens the possibility to specify temporal-logic properties at the UML level and to verify them at the XTG level using the PMC model checker.

1 Introduction

The Unified Modeling Language (UML) is the object modeling standard [1] that provides a set of diagrams for system specification from static and dynamic perspectives [6]. Nowadays, it is becoming more and more custom for designers to use formal methods during the design. The formal methods allow to verify a system specification with respect to its property specification. The system specification in UML can be formalized [4], however, the property specification is limited by the logic of the Object Constraint Language (OCL) [6] which does not allow properties of computational paths, reachability, etc. to be specified [2].

This paper shows our approach to specification of systems and properties in UML. In section 2 we consider a tuple of UML class, object and statechart diagrams as a new input language of the Prototype Model Checker (PMC) being under development in Delft University of technology [7]. PMC is intended to verify system specification with respect to properties represented in a variant of Time Computation Tree Logic (TCTL). However, the level of the original system specification language of PMC, namely Extended Timed Graphs (XTG), is too low for the specification purpose. The specification language of the acceptable level is UML. In section 3 we represent the transformation of the UML specification into XTG. In section 4 we conclude about the results of the transformation that allows to specify system properties by TCTL at the UML level and to verify them by means of the PMC.

2 System Specification. Property Specification

1. *We have chosen three kinds of UML diagrams to specify a system.*
– Class and object diagrams define a static view of the system. A class diagram specifies sets of classes Cl_S with their attributes At_S and operations Op_S. An object diagram defines a current set of class instances.

To enable the measurement of time we introduce clock attributes of special *DenseTime* type. A clock attribute can be reset to a nonnegative real value. After resetting the value of the clock attribute continuously increases.
– A UML statechart diagram addresses a dynamic view of the system. All labels of the statechart use names that are defined by the UML class and object diagrams [3].

We define a UML statechart as a tuple $SchD = (S, S_0, Rl)$, where
– S is a tree of states. The states are depicted in the statechart diagrams by boxes with round corners. There are states of three types in this tree: AND, XOR and *Simple*. AND and XOR nodes are hierarchical states. One of them is a root of the tree of states. Nodes $A_1, \dots A_n$ of a hierarchical state C are drawn inside of C. An AND-state is divided by dotted lines to put other states between such lines. A simple state does not contain other states. In general, a state s is marked by $(Name_s, History_s, In_s, Out_s)$ labels. The labels $History_s$, In_s, Out_s can be empty.
– $S_0 \subseteq S$ is a set of initial states.
– Rl is a set of relations among states $Rl = \{TR, Conn, Synch\}$.
  – TR is a set of transitions that are represented by labeled arrows.
    $TR = \{(s_i, s_j, e, g, a) | s_i, s_j \in S;\ e \in Op_S,$
    $g \in \ Boolean\ Expression(At_S, Op_S), a \in Op_S\}$.
  – $Conn = \{(I, O) |\, I \subset S, O \subset S, \}$. The relation is represented by a black rectangle (fork-join connector) and by arrows that are directed from each state $I_i \in I$ to the connector and from the connector to each state $O_j \in O$.
  – $Synch = \{(I, O) |\, I \subset S, O \subset S\}$. The relation is drawn by two black rectangles (fork and join connectors) and by a circle of a synch-state.

A transition semantics of a UML-statechart, in general, was defined in [4] as a computational tree. A node of this tree is a vector $n = (s, v, q)$, where $s \subset S$ of the $SchD$, v is a tuple of values of all attributes of objects from the object diagram, q is a queue of operation calls from the $SchD$.
2. *To enable the specification of properties* of computational sub-trees, we define specification classes. Specification classes are stereotyped. They can be related to a set of traditional classes. Each stereotype of specification has its own intuitive name and a formal representation by a parameterized TCTL formula [3]. The TCTL variant that we use [3,7] contains location predicates and reset quantifiers over variables.

3 Transformation Rules

We transform a tuple of UML class, object and statechart diagrams into XTG, the original language of the Prototype Model Checker.

3.1 XTG

XTG is a formalism for describing real-time systems.

An XTG is a tuple $G = (V, L, l_0, T)$, *where*
– V is a finite set of variables of the following

$$DataType = \{Integer, Enumeration, Real, DenseTime\}.$$

$DenseTime$ is a type for representing clocks (as nonnegative real that increases continuously).
– L is a finite set of locations (nodes). $l_0 \in L$ is an initial location;
– T is a finite set of transitions (arrows with labels). $T = \{(l_i, l_j, c, up, ur)\}$, where $l_i, l_j \in L$, $c \in BooleanExpression(V)$, $up \in ValueAssignment$ of the variables V, $ur \in \{Urgent, UnUrgent\}$. An urgent transition is represented by an arrow with a black dot.

A state in the time computation tree semantics of the XTG is defined [7] by a location and the values of variables in the location (l, ρ). A transition from a state (l, ρ) is enabled if $c(\rho) = True$ after the substitution of the variable values. Time can pass in state as long as its invariant is satisfied and no urgent transitions are enabled. Time is not allowed to progress while a transition marked as urgent is enabled. The parallel composition of XTG is defined [7] by a synchronization mechanism that is based on a form of value passing CCS [5]. A synchronization is a pair of labels specifying that the transition marked by *syn*! in an XTG is executed simultaneously with a transition marked by *syn*? in another XTG.

3.2 Transformation Steps

We make the transformation of the UML specification into XTG in three steps.

- First, we represent the hierarchical states from the statecharts introducing the CCS synchronization. The result of this transformation we name the flat statecharts. A *flat statechart* is a UML statechart which does not contain hierarchical states and uses the CCS synchronization mechanism. The parallel composition of two flat statecharts is a flat statechart such that the set of its transitions is defined by rules of XTG parallel composition.
- Second, we transform the flat statecharts defining the *run to completion* [6] semantics of operation calls and semantics of signals using the synchronization.
- The last step means the transformation of fork-join connectors and synch states of the flat statescharts to XTG.

From Hierarchical Statecharts to Flat Statecharts

XOR state with the History mark. The transition to the XOR state (fig.1) means the transition to the initial state a_1 among substates a_1, a_2. The transition from the XOR state c_1 means the transition from the current state a_i. Only one substate from the a-set can be the current substate. The history label means

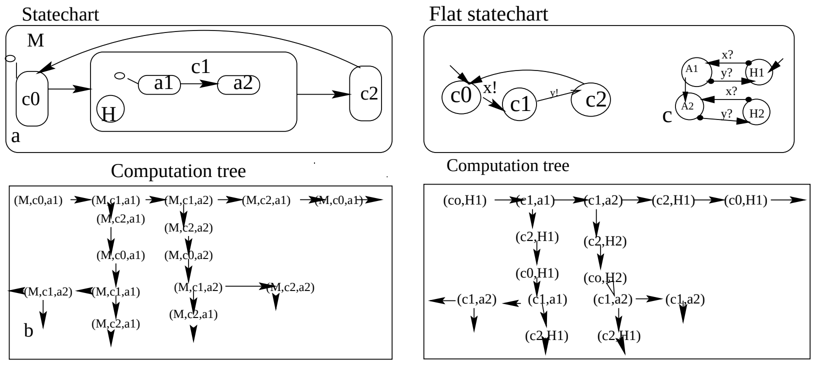


Fig. 1. XOR states

memorizing of the current substate of the XOR state and starting the next computation inside of the XOR state from this substate. The semantics is shown by the computation tree (fig.1b).

We represent the statechart (fig.1) by two XTG: the external graph (c_0, c_1, c_2) and internal one (a_1, a_2). The history label means that there is a history state H_i for each a_i. States c_0, H_1 are initial for the system. The transition to the XOR state is replaced by two synchronous transitions: $(c_0, c_1, x!)$ and $(H_i, a_i, x?)$. The value of i depends on the last active state of the internal graph. The transition from the XOR state is replaced by pair of synchronous states $(c_1, c_2, y!)$ and $(a_i, H_i, y?)$.

AND state. The transition to the AND state C(fig.2a) corresponds to transitions to both initial states of substatecharts of A and B. The arrow from AND state C means the transitions from the current states of A and B. The transition from each state of A, B is possible. The corresponding computation tree is shown in fig.2b. In the flat statechart representation there are three sub-graphs (fig. 2c). There is a projection of the computation tree (fig. 2d) of the flat statechart to the computation tree of the statechart (fig.2b). In this projection one transition to AND state $(M, X, A_1, B_1) \rightarrow (M, C, A_1, B_1)$ is modeled by two transitions in the computation tree of flat statechart $(X, A_0, B_0) \rightarrow (X_1, A_1, B_0) \rightarrow (C, A_1, B_1)$. A transition from AND state is modeled by a subtree: for example, $(M, C, A_2, B_2) \rightarrow (M, Y, A_1, B_1)$ (fig. 2b) is represented by $(C, A_2, B_1) \rightarrow (Y_1, A_0, B_1) \rightarrow (Y, A_0, B_0)$ and $(C, A_2, B_1) \rightarrow (Y_1, A_2, B_0) \rightarrow (Y, A_0, B_0)$. *AND*

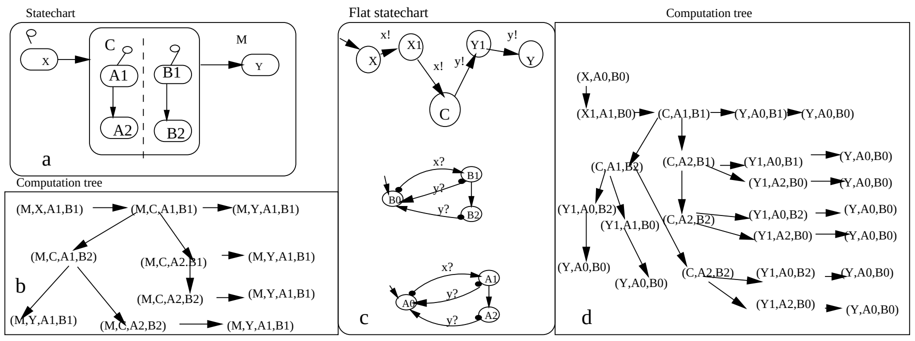


Fig. 2. AND states

state with the history label are transformed to flat statecharts using the combination of rules given in two previous cases.

Transformation of Operation Calls and Signals

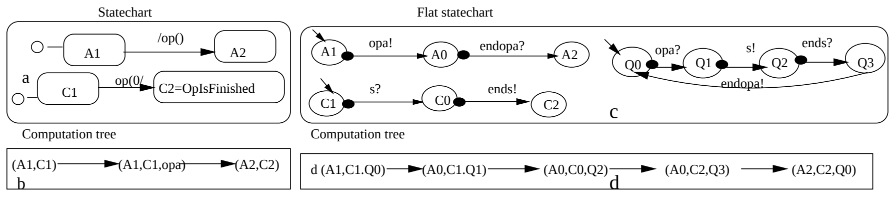


Fig. 3. Operation in statecharts

Let an *operation op* be called by the object A and be fulfilled by the object C (fig.3a). If the operation call is shown in UML statechart, it means that the operation will run to completion. The corresponding computation tree is represented in the fig.3b. The operation call of the object A is shown by the label *opa*. The time between the operation call and the operation end [6] is the subject of the specification in real-time systems and it can be shown by updates of clock attributes at the moment of the operation call. When an operation is called by several classes then there is a time between the operation call and the moment of the begin of the operation. To represent the semantics of the operation calls we extend graphs of objects A, C by states A_0, C_0 and use an additional flat statechart Q with four states to model each operation call (fig.3c.) The internal state of this graph Q_1 allows to wait for the moment of the begin of the operation. This moment is shown by synchronization s with the graph C. When the operation has been fulfilled, graph Q is synchronized by end_{opA} with graph A. The computation tree of the flat statechart is shown in fig.3d. There is a projection of this tree to the computation tree of the statechart: three transitions $(A_0, C_1, Q_1) \to (A_0, C_0, Q_2) \to (A_0, C_2, Q_3) \to (A_2, C_2.Q_0)$ in the computation tree of the flat statechart correspond to one transition $(A_1, C_1, opa) \to (A_1, C_2)$ in the computation tree of the statechart.

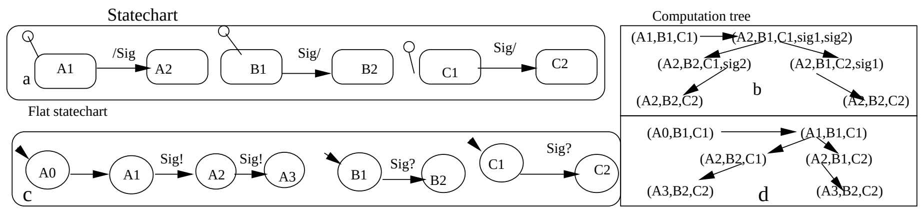


Fig. 4. Signal in statecharts

Let a *signal* be sent to two objects fig.4a. The moment of sending is a subject of specification in real time systems. The reactions to the signal can come in different moments. To represent the behaviour of a signal by flat statecharts (fig.4 b) we extend the set of states to fix the moment of the signal sending and to send 2 synchronizations $Sig!$. We can see in fig.4b,d that the computation tree of the flat statecharts and the computation tree of the statechart are equal.

Transformation of Flat Statecharts
Assume now that all XOR, AND states, operation calls and signals have been transformed and a flat statechart has represented a system specification.

In fig.5 there is a statechart with a *fork-join* connector and its XTG representation. The semantics of the join-fork statechart is the following. First, all

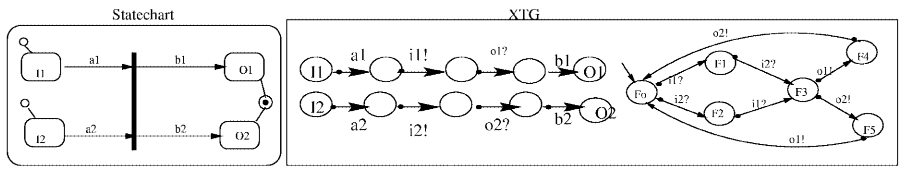


Fig. 5. Fork-Join connector

transitions to the connector have to be finished. The order of the transitions is nondeterministic. When all transitions to connector have been finished then the transitions from the connector are enabled. The order of the transitions is nondeterministic.

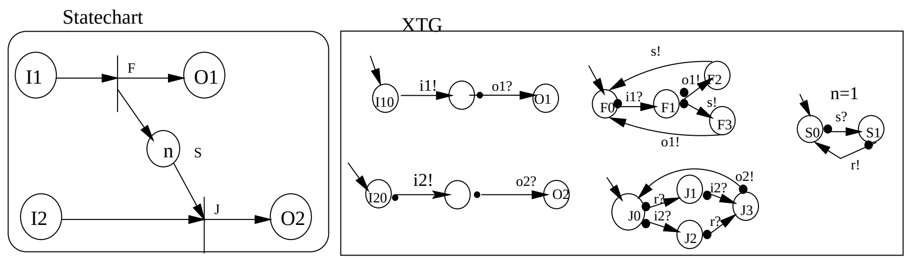


Fig. 6. Synch state

Synch state. The graph that is defined by means of *Synch*-relation is a statechart fig. 6. The semantics of this statechart uses the semantics of connectors and the synch state. The last one guarantees that one region of statechart leaves a particular state before another region can enter a particular state. If a synch state is marked by number 1 then one input to synch state corresponds to one output from it. This case is shown in the fig. 6. If a synch is marked by number n then one input to synch-state can correspond to n outputs from it. An unlimited synch-state is marked by symbol $*$.

In and *Out* labels of the state mean that all edges to the state are marked by *In* and all edges from the state are marked by *Out*.

4 Conclusion

Analyzing the transformation rules of our UML specification into the input language XTG of the PMC toolset we can conclude that there is a projection of the XTG computation tree into the computation tree of the corresponding UML specification. So, the properties of computational sub-trees specified at the UML level are visible at the XTG level. The TCTL variant that we use [3,7] contains state predicates and reset quantifiers over clocks. For example, we would like to specify a *Deadline* for the state y for the statechart with an AND-state (fig. 2), i.e. if we are situated in state x of the class X we should achieve state y till the deadline T. The stereotype *Deadline* has the clock attribute t , the parameters: deadline T, two state predicates $X@x$, $X@y$ and the following TCTL formula: $AG(X@x \Rightarrow (t := 0).(AF(X@y \wedge t \leq T)))$.

The PMC tool is able to verify temporal properties of systems with unlimited sets of traces. The combination of the model checker power and the UML specification power allows to apply formal methods into every day practice of design.

References

1. G. Booch, J. Rubaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Amsterdam, 1999.

2. B. P. Douglass. *Real-Time UML. Developing Efficient Objects for Embedded Systems.* Addison-Wesley, 1998.
3. E.E. Roubtsova and J.van Katwijk and W.J. Toetenel and C. Pronk and R.C.M.de Rooij. The Specification of Real-Time Systems in UML. *MCTS2000*, http://www.elsevier.nl/locate/entcs/volume39.html , 2000.
4. J. Lilius and I.P.Palor. Formalising UML StateMachines for Model Checking. *UML'99. Beyond the Standard, LNCS 1723*, pages 430–445, 1999.
5. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
6. OMG. *Unified Modeling Language Specification v.1.3.* ad/99-06-10, http://www.rational.com/uml/resources/documentation/index.jsp, June 1999.
7. R.F. Lutje Spelberg and W.J. Toetenel and M. Ammerlaan. Partition Refinement In Real-Time Model Checking. *Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS* , 1486:143–157, 1998.

A Systematic Approach towards Object-Based Petri Net Formalisms*

Berndt Farwer[1] and Irina Lomazova[2]

[1] University of Hamburg, Department of Computer Science,
farwer@informatik.uni-hamburg.de
[2] Program Systems Institute of Russian Academy of Science,
irina@univ.botik.ru

Abstract. The paper aims at establishing the semantical background for extending Petri net formalisms with an object-oriented approach by bringing together Nested Petri Nets (NP-nets) of Lomazova and Linear Logic Petri nets (LLPNs) of Farwer. An introductory example shows the capabilities of these formalisms and motivates the proposed inter-encoding of two-level NP-nets and LLPNs. A conservative extension to the Linear Logic calculus of Girard is proposed – Distributed Linear Logic. This extension of Linear Logic gives a natural semantical background for multi-level arbitrary token nets.

1 Introduction

Modularisation and the object-oriented programming paradigm have proved to be facultative for the development of large-scale systems, for instance in flexible manufacturing, telecommunications, and workflow systems. These efficient approaches to systems engineering stimulated researchers in the Petri net community to develop new flavours of Petri nets, such as LOOPN of C. Lakos [6] and OPN of R. Valk [10], which incorporate object-oriented aspects into Petri net models (see also the survey [11], referring to a large collection of techniques and tools for combining Petri nets and object-oriented concepts). The design of these new net formalisms has led to the problem that only very few results from the traditional theory of Petri nets are preserved and thus the spirit of the Petri net model is altered considerably. Moreover, many of the object-oriented extensions of Petri nets are made *ad hoc* and lack a formal theoretical background.

The formalisms studied in the present paper take a more systematic view of objects in Petri nets in order to allow the application of standard Petri net techniques, and hence they do not follow the object-orientation known from programming languages in every aspect. Their study aims to give some insight into how objects can be integrated into Petri nets in a way that does not violate the formal basis of the initial model.

* This research was partly supported by the Russian Foundation for Basic Research, Project No. 99-01-00309

We consider two independently proposed extensions of ordinary Petri net formalisms, both of which utilise tokens representing dynamic objects. The idea of supplying net tokens with their own net structure and behaviour is due to R. Valk [10].

In Nested Petri nets (NP-nets) [7] tokens themselves are allowed to be nets. In contrast to Object Petri nets of Valk, where an object net token may be in some sense distributed over a system net, in NP-nets a net token is located in one place (w.r.t. a given marking). This property facilitates defining formal semantics for NP-nets and makes possible a natural generalisation of this model to multi-level and even recursive cases [8]. The main motivation for introducing NP-nets was to define an object-oriented extension of Petri nets, which would have a clear and rigorous semantics, but still be weaker than Turing machines and maintain such merits of Petri net models as decidability of some crucial verification problems. It was stated in [9], that while reachability and boundedness are undecidable for NP-nets, some other important problems, namely termination and coverability, remain decidable.

Linear Logic Petri nets (LLPNs) have been introduced as a means for giving purely logical semantics to object-based Petri net formalisms. The basic property that makes Linear Logic, introduced in 1989 by Girard [5], especially well-suited for this task is the resource sensitivity of this logic . The nature of Linear Logic predestines it also for the specification of Petri nets with dynamic structure [2]. A Linear Logic Petri net is a high-level Petri net that has Linear logic formulae as its tokens. The token formulae can be restricted to a fragment of Linear Logic to maintain decidability. Arcs have multisets of variables as inscriptions and the transitions are guarded by sets of Linear Logic sequents that are required to be derivable in the applicable sequent calculus for the transition to occur. Then in [2] it was shown, that two-level NP-nets also allow a natural Linear Logic representation, and thus have a very close relation to LLPNs.

Linear Logic of Girard has proved to be a natural semantic framework for ordinary Petri nets, such that the reachability of a certain marking in the net corresponds to the derivability of the associated sequent formula in a fragment $\mathbf{ILL}_{\mathrm{PN}}$ of the intuitionistic Linear Logic sequent calculus. In this paper we propose an extension of Linear Logic to Distributed Linear Logic, which allows, in particular, giving Linear Logic semantics to multi-level object nets. The main difference from classical Linear Logic will be in that resources, represented by formulae, will be distributed (belong to some owners or be distributed in space), so that two resources located in different places (or belonging to different owners), generally speaking, cannot be used together. To express this we propose the use of the special binary operation P ("possesses"). We write $A(\phi)$ for $P(A, \phi)$, where A is an atomic symbol, representing an owner or location, and ϕ is a Linear Logic formula (possibly including the "possesses" operation).

There is a natural interpretation of the "possesses" operation, which continues Girard's example from [5] about possessing a dollar and buying a box of cigarettes. Let D be a dollar and C be a pack of cigarettes. Then $A(D)$ designates that person A owns a dollar; $A(!(D \multimap C))$ means, that A has an ability to obtain cigarettes for a dollar (e.g. A is not a child). Further in this setting $A(D) \multimap B(D)$ means that A can pass his dollar to B, and $A(x) \multimap B(x)$ (implicitly universally quantified for x) means that A is ready to give B everything he or she owns.

The paper is organised as follows. Section 2 gives a brief introduction to the main Linear Logic connectives leading to the encodings of traditional Petri nets, coloured Petri nets (Section 3) and to the definition of Linear Logic Petri nets. Section 4 contains an example of modelling by NP-nets and LLPNs, showing some connections between the two formalisms. Section 5 is devoted to a formal translation of NP-nets into LLPNs and vice versa and contains results presented in [3]. Here the NP-nets are assumed to have only bounded element nets. In Section 6 we introduce an extension of Linear Logic by the "possesses" operator and use it for the Linear Logic representation of multi-level NP-nets. Section 7 gives some concluding remarks.

2 Linear Logic Encoding of Petri Nets

Girard's Linear Logic [5] differs from classical logic in that the structural rules of weakening and contraction in a Gentzen-style sequent calculus are restricted to special versions where the main formulae are in the scope of a special modality called *of course*, which is needed to retain the expressiveness of classical logic. This modality is used to make a formula reusable – a typical classical property, which is also of utmost importance for the modelling of Petri net transitions.

Every formula is treated a a resource in Linear Logic. Thus, a derivation using an implication and its premises actually consumes these and produces the respective consequences in the form of new instances of resources of an appropriate type.

The lack of structural rules results in two fragments of Linear Logic, called the multiplicative and additive fragment. Each fragment has its own kind of conjunction and disjunction. The most interesting fragment for the encoding of Petri nets is the multiplicative fragment, because it is resource sensitive. The additive fragment behaves essentially like in classical logic and is less interesting for our goal.

The multiplicative conjunction ($\otimes$, *tensor* or *times*) is used to accumulate resources. It is essential to understand that unlike in the case of the classical conjunction where $A \wedge A = A$, this equation does not hold for the multiplicative conjunction, i.e. $A \otimes A \neq A$.

The behaviour of a Petri net is determined by the consumption and production of tokens when firing transitions. Modelling this as a Linear Logic formula, one has to remember that a linear implication is also treated as a resource unless it is preceded by the modality *of course*. A transition with input places A and B, output places B and C and an arc weight of one on all arcs except the input arc from A, which has weight two (see Figure 1), can be represented by $!(A \otimes A \otimes B \multimap B \otimes C)$. The use of the modality is essential, because without it the implication could be used only once in a derivation, but the transition can actually fire each time its input places contain sufficiently many tokens.

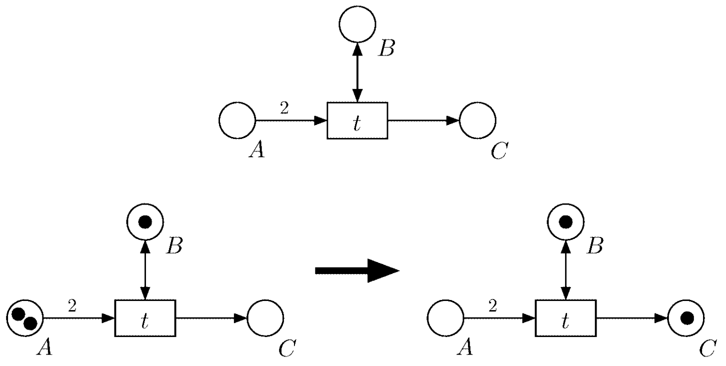


Fig. 1. A Petri net transition

For example,

$$A \otimes A \otimes B \otimes !(A \otimes A \otimes B \multimap B \otimes C) \ \vdash \ B \otimes C \otimes !(A \otimes A \otimes B \multimap B \otimes C)$$

is provable, but

$$A \otimes A \otimes B \otimes (A \otimes A \otimes B \multimap B \otimes C) \ \vdash \ B \otimes C \otimes (A \otimes A \otimes B \multimap B \otimes C)$$

is not, since the implicational subformula would have to be consumed in the derivation.

Remark 1. In this paper we assume that the modality ! has stronger binding preference than the conjunction, which in turn has stronger binding preference than the implication. We use this rule of preference to keep low the amount of parentheses in formulae, making them more readable.

3 Modelling Coloured Petri Nets by Linear Logic

In this section we give a Linear Logic representation of Coloured Petri nets with a finite set of colours, a finite set of atomic individual tokens of different colours, and without guards. We extend the language $L(\mathbf{ILL}_{\mathrm{PN}})$ of the intuitionistic Linear Logic to the language $L(\mathbf{DILL}_{\mathrm{PN}})$ by adding the binary operation P (for possesses) in the following way.

On the syntactical level we have:

- a finite set of atom symbols $A, B, \ldots$ (corresponding to places);
- a finite set of atom symbols $a, b, \ldots$ (corresponding to token colours);

- variables $x, y, \ldots$;
- the binary logical operations $\otimes$, $\multimap$;
- the modality "of course" !;
- the binary operation symbol P (for possess).

Formulae (terms) are defined as follows:

- token atoms $a, b, \ldots$ and variables $x, y, \ldots$ are terms;
- if A is a place atom, ϕ — a term, then $P(A, \phi)$ is a term. We write $A(\phi)$ as a shorthand for $P(A, \phi)$;
- if ϕ and ψ are terms, then $\phi \otimes \psi$, $\phi \multimap \psi$ and $!\phi$ are terms.

As usual for sequent calculi we define a sequent formula as an expression of the form $\Gamma \vdash \phi$, where Γ is a sequence of terms and ϕ is a term.

Variables are interpreted as terms. The $\mathbf{DILL}_{\mathrm{PN}}$ calculus is obtained by adding to $\mathbf{ILL}_{\mathrm{PN}}$ the following rule for substituting a term for a variable:

$$\frac{\Gamma \vdash \phi}{\Gamma[\psi/x] \vdash \phi[\psi/x]}$$

Definition 1 (canonical formula for CPN). *The* extended canonical formula *of a coloured Petri net* $\mathcal{N} = \langle P, T, F, \mathcal{C}, C, G, V, \mathbf{m_0} \rangle$ *is defined by the tensor product of the following factors. W.l.o.g., assume that each arc is carrying either a variable, or a constant (then arc expressions of the form* $x + 2y$ *can be represented by multiple arcs):*

- *For each transition* $t \in T$ *construct the factor*

$$!\left(\bigotimes_{p \in {}^\bullet t} p(W_\otimes(p,t)) \multimap \bigotimes_{q \in t^\bullet} q(W_\otimes(t,q)) \right),$$

- *Construct for the current marking* $\mathbf{m}$ *and all places* $p \in P$ *and tokens* a *with* $a \in \mathbf{m}(p)$ *the formulae* $p(a)$. *Thus, for the complete marking we get*

$$\bigotimes_{\substack{p \in P \\ a \in \mathbf{m}(p)}} p(a),$$

  where multiple occurrences of tokens contribute to multiple occurrences of factors in the tensor product.

Thus, for example, the formula

$$!(A(x) \otimes B(y) \multimap C(x) \otimes C(x))$$

corresponds to the transition shown in Figure 2.

It can be shown that the canonical formula $\Psi_{\mathbf{DILL_{PN}}^{EN}}(\langle \mathcal{N}, \mathbf{m} \rangle)$ for a marked CPN $\langle \mathcal{N}, \mathbf{m} \rangle$ is constructed in such a way that

$$\Psi_{\mathbf{DILL_{PN}}^{EN}}(\langle \mathcal{N}, \mathbf{m} \rangle) \vdash \Psi_{\mathbf{DILL_{PN}}^{EN}}(\langle \mathcal{N}, \mathbf{m}' \rangle)$$

is derivable iff the marking $\mathbf{m}'$ is reachable in the marked net $\langle \mathcal{N}, \mathbf{m} \rangle$.

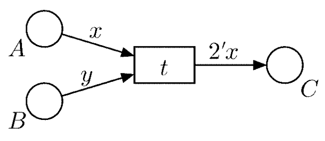


Fig. 2. A coloured Petri net transition

4 Object-Based Modelling with NP-Nets and LLPNs

In this section we discuss the encoding of net tokens, i.e. tokens in object Petri nets, that are themselves Petri nets. As before, in the Linear Logic encoding of a current marking, a separate factor of the form $P(\phi)$, where ϕ is an encoding of the corresponding net token, will represent a token occurrence in the place P. If there were no synchronisation, we would encode transitions as before and only add to $\mathbf{DILL}_{\mathrm{PN}}$ the following rule for representing autonomous behaviour of net tokens:

$$\frac{F \ \vdash \ G}{A(F) \ \vdash \ A(G)}$$

To deal with horizontal and vertical synchronization, as they appear in NP-nets, we have to employ a more subtle transition encoding, which will be introduced after some remarks on nested Petri nets in general.

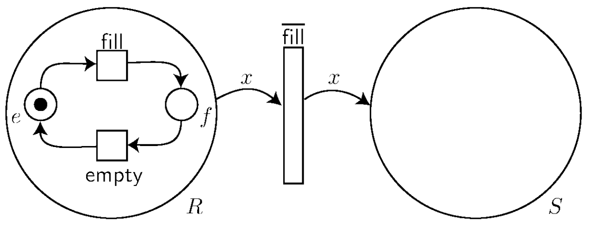


Fig. 3. Example of NP-net

In NP-nets tokens are nets. A behaviour of a NP-net includes three kinds of steps. An autonomous step is a step in a net in some level, which may “move”, “generate”, or “remove” its elements (tokens), but does not change their inner states. Thus, in autonomous steps an inner structure of tokens is not taken into account. There are also two kinds of synchronisation steps. Horizontal synchronisation means simultaneous firing of two element nets, located in the same place of a system net. Vertical synchronisation means simultaneous firing of a system net together with its elements “involved” in this firing. Formal definitions of NP-nets can be found in [7,8].

Figure 3 shows an example of a NP-net, where a system net has two places R and S and one transition marked by $\overline{\mathsf{fill}}$. In the initial marking it has one element net (with two places e and f and two transitions marked by fill and empty) in R. Here only a vertical synchronisation step via synchronisation label fill is possible. It moves the element net token to S and simultaneously changes its inner marking to f.

An equivalent Linear Logic Petri net is shown in Figure 4.

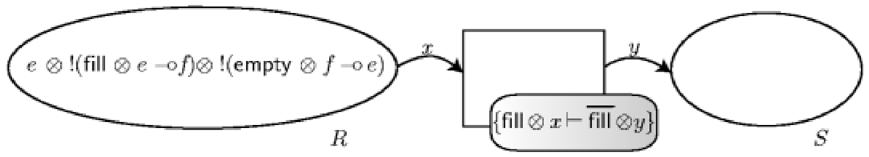

$e \otimes !(\mathsf{fill} \otimes e \multimap f) \otimes !(\mathsf{empty} \otimes f \multimap e)$
x
y
$\{\mathsf{fill} \otimes x \vdash \overline{\mathsf{fill}} \otimes y\}$
R
S

Fig. 4. LLPN equivalent to the NP-net from Figure 3

Linear Logic Petri nets (LLPNs) are high-level Petri nets that have Linear Logic formulae as tokens. Arcs are inscribed by multisets of variables and transitions are guarded by *guard sets* of Linear Logic sequents that include the neighbouring arc variables. A transition is enabled if there are sufficient token formulae in the respective input places and an appropriate binding, such that all guard sequents of one guard set are derivable in the underlying calculus. The formulae bound to the input variables are removed from their respective places and new formulae are placed on the output places according to the binding, the derivation, and the output arc inscriptions.

Autonomous actions can take place in two different ways: Firstly, the environment net can have transitions that do not force the token formulae residing in its input places to undergo any derivation upon firing the transition. In this case the effect of firing the transition can be described as simply redistributing some token formulae without changing them. Secondly, the token formulae may evolve at any time according to the derivation rules of the underlying calculus, but remaining in the same place of the net, i.e. a formula α residing in some place may evolve to some other formula β without the occurrence of any transition in the Petri net, provided $\alpha \vdash \beta$ is provable in the underlying calculus. These two kinds of autonomous action correspond to the autonomous firing of an environment net transition (transport) and the autonomous firing of an element net transition in a nested Petri net.

The token formulae of a Linear Logic Petri net can be used to encode the element nets of nested Petri nets. In the following sections we develop an encoding of nested Petri nets a formulae, such that a Linear Logic formula represents a complete nested Petri net, thereby establishing the foundation for a simulation of multi-level nested Petri nets by Linear Logic Petri nets. Modelling horizontal and vertical synchronisation is achieved by using *synchronisation tags*, i.e. propositional symbols that restrict possible derivations of the token formulae and enforce the synchronisation between some derivation steps with the firing of a transition in the environment net.

The reader is referred to [1,4] for an introduction to LLPNs. A discussion of further issues, especially dealing with dynamic modifications of object net structures can be found in [2].

5 Inter-representability of Two-Level NP-Nets and LLPNs

This section gives a brief sketch of a formal translation of two-level NP-nets into LLPNs and vice versa. It contains results presented in [3] and restricts the class of NP-nets to those that have only bounded element nets. For a translation of multi-level NP-nets refer to the extension of the Linear Logic calculus proposed in section 6.

The main issue of any translation of an object Petri net formalism into a LLPN framework is how to deal with synchronisation. Instead of giving the complete translation of a given NP-net into its corresponding LLPN (which can be found in [3]), we sketch the main idea of synchronisation within LLPNs. The formal translation then follows as an obvious consequence.

For each pair of system and element net transitions t_s and t_e with adjacent vertical synchronisation labels ℓ and $\overline{\ell}$ we define two new unique propositional symbols with the same name as the labels. W.l.o.g. assume these labels to be disjoint with any place names used in either net. We call the pair of labels *message handles* or *synchronisation labels* for the synchronisation of t_s and t_e.

Let, for example, $!(A \multimap B)$ be a partial canonical formula of an element net $\mathcal{N}$ residing in place p of a Linear Logic Petri net $\mathcal{LLPN}$. The synchronisation of the system net transition t_s with a transition t_e in the element net via message handles $(l, \overline{\ell})$ is represented by the token formula

$$!(l \otimes A \multimap \overline{\ell} \otimes B)$$

and by a transition in $\mathcal{LLPN}$ with the guard function

$$G(t) = \{\ell \otimes x \quad \vdash \quad \overline{\ell} \otimes y\}.$$

Here the message handles are used to ensure that the derivation step, which takes place during the firing of the system net transition of the LLPN, really uses the Linear Logic implication representing the element net transition t_e.

Thus, a synchronisation of a firing in the system net with the derivation step outlined above coincides with the vertical synchronisation step of the simulated NP-net. Horizontal synchronisation can be simulated analogously.

The simulation relies on the boundedness of the NP-net, since it uses a propositional calculus for the encoding. A generalisation of the calculus to overcome this restriction is discussed in the remainder of the paper.

Theorem 1. *For every unary elementary NP-net NPN without constants in arc inscriptions, there exists a canonical Linear Logic Petri net $\mathcal{LLPN}_{NPN}$, such that*

– *the token formula in $\mathcal{LLPN}_{NPN}$ is the canonical representation of the element net of NPN,*
– *there is a bijection between the transitions of NPN and $\mathcal{LLPN}_{NPN}$, which generates the bijection between occurrence sequences of the system net in NPN and occurrence sequences of $\mathcal{LLPN}_{NPN}$.*

On the contrary, the simulation of LLPNs by NP-nets is possible only for a fairly restricted class of LLPN, since the definition of LLPNs is much more general. It allows variables to occur multiply in input and output arc inscriptions and the use of transition guards. [3] gives some conditions for the possibility of a NP-net simulation for LLPNs.

6 Extending the Linear Logic Calculus

Linear Logic of Girard due to its resource sensitivity has a close resemblance to Petri nets: it has connectives that can handle resources in the same manner as ordinary Petri nets do. A Linear Logic representation of high-level Petri nets, such as e.g. coloured Petri nets of K. Jensen, can be achieved by simulating the "unfolding" of places and transitions (cf. [1]), when for each token colour a new copy of the place is created (technically, the set of places indexed by the possible colours represents the new set of propositional atom symbols for the encoding).

To obtain a straightforward Linear Logic encoding of high-level Petri nets the intuitionistic Linear Logic calculus $\mathbf{ILL}_{\mathrm{PN}}$, used so far to encode Petri nets, can be extended from propositional to predicate level. Now formulae may contain variables, and all variables are supposed to be universally quantified. Then we introduce a special possess operation P for encoding that a place A contains a token x. Note, that components of NP-nets are high-level nets with net tokens, so the formula of the form $P(A, \phi)$ will designate, that a net token with encoding ϕ belongs to the place A.

More formally, we extend the language $L(\mathbf{ILL}_{\mathrm{PN}})$ of intuitionistic Linear Logic to the language $L(\mathbf{DILL}_{\mathrm{PN}})$ by adding the binary operation $P(\cdot,\cdot) \subseteq \mathcal{A} \times L(\mathbf{DILL}_{\mathrm{PN}})$, where $\mathcal{A}$ is a set of atomic symbols, representing owners or locations, in the way described in the previous section for coloured Petri nets.

Now we can define the Linear Logic encoding of a NP-net by its canonical formula. Here, in the Linear Logic encoding of a current marking a separate factor of the form $A(\phi)$, where ϕ is an encoding of the corresponding net token, will represent a token occurrence in the place A. Analogously to the LLPN representation of NP-nets, given in the previous section, new propositional symbols with the same names as labels are used for encoding horizontal and vertical synchronisations. The approach taken here is completely symmetrical in the sense that each encoded net is provided with the facilities to participate in a horizontal synchronisation, or in a vertical synchronisation in either of the two possible parts, i.e. it can synchronise with a net on a lower or upper level.

Remark 2. Note, that we have to take precautions to keep the sets of variable names used in different net token instances involved in a synchronisation step disjoint in order to avoid ambiguities. In the following, we use $W_{\otimes}(p,t)$ and $W_{\otimes}(t,p)$ to denote the tensor product $\bigotimes_{x\in W(p,t)} p(x)$ and $\bigotimes_{x\in W(p,t)} p(x)$ respectively. W.l.o.g. we assume all variables used in arc inscriptions form a set $\mathcal{V}$, such that for all $x \in \mathcal{V}$ we have $\bar{x} \notin \mathcal{V}$. Then, by $\overline{W}_{\otimes}(p,t)$ we denote the product $\bigotimes_{x\in W(p,t)} p(\bar{x})$ (analogously for $\overline{W}_{\otimes}(t,p)$) .

A nested Petri net consists of the following components: $\overline{\mathcal{N}} = \{\mathcal{N}_0, \ldots, \mathcal{N}_k\}$ is a finite set of net components, such that all place and transition names of net components in $\overline{\mathcal{N}}$ are pairwise disjoint; A_a is a finite set of atomic tokens; $\mathcal{I}$ is an interpretation of constants in $\overline{\mathcal{N}}$ over A_a and $\mathrm{A}_n(\overline{\mathcal{N}}, \mathrm{A}_a)$; $\mathcal{N}_0$ is a distinguished net component, called the *system net*; $\mathbf{m}$ is a feasible marking of the system net $\mathcal{N}_0$ over $\overline{\mathcal{N}}$ and A_a, called *initial marking* of a NP-net $\langle P, T, W, \mathbf{m}\rangle$.

Note also, that the definition of NP-net has some strict conditions on the use of variables for arc inscriptions: No variable may occur more than once on any input arc, and a variable occurring on one input arc may not occur on another input arc of the same transition. All variables occurring on output arcs of a transition must also occur on an input arc, i.e. the variables on output arcs of a transition are a subset of the variables from it's input arcs.

To each horizontal or vertical synchronisation label ℓ there exists an adjacent label $\bar{\ell}$ that is needed in an appropriate marking for a synchronous synchronisation step to occur. By definition we have $\bar{\bar{\ell}} := \ell$.

The previous remarks are important for an understanding of the definition of canonical formula given below. Variables named $\Gamma, \Gamma_x, \alpha, \alpha_x, \ldots$ are assumed to be new – otherwise unused – variables. The subterms Γ, Γ_x, and Γ' are used as variables for the remainder of any nets that interact in some kind of synchronisation, i.e. the part of the net description that is left untouched by firing the transitions involved simultaneously. For example, in the NP-net of Figure 3 only the token net transition marked with the label fill can synchronise with the system net transition marked $\overline{\mathsf{fill}}$. Thus, for the encoding only these parts of the net structure are important for the synchronisation. The remainder (in this case the transition marked empty) must not be altered in the synchronisation step. In the simulating derivation this is accomplished by assigning the remainder to the variable Γ, which is passed through to the consequence of the linear implication. Hence, the resource is not used and is untouched by the derivation just like the remainder of the net.

Definition 2 (canonical formula for NP-net). *The* extended canonical formula *of a NP-net* $(\overline{\mathcal{N}}, \mathrm{A}_a, \mathcal{I}, \mathcal{N}_0, \mathbf{m})$ *is defined by the tensor product of the following factors. W.l.o.g., assume that each arc is inscribed with either a variable or a constant, and let P denote the union of all place sets, T the union of all transition sets, and L the union of all label sets.*

– *For each autonomous transition $t \in T$ construct the factor*

$$!\left(\bigotimes_{p\in {}^{\bullet}t} p(W_{\otimes}(p,t)) \multimap \bigotimes_{q\in t^{\bullet}} q(W_{\otimes}(t,q))\right), \tag{1}$$

– *For each transition $t \in T$ with a vertical synchronisation label ℓ such that $\forall p \in {}^\bullet t \, . \, W(p,t) \notin \mathbb{N}$ holds (i.e. all input places on the next lower level must hold net tokens), construct the factor*

$$! \left(\bigotimes_{p \in {}^\bullet t} \bigotimes_{x \in W(p,t)} p(\Gamma_x \otimes \alpha_x \otimes !(\bar{\ell} \otimes \alpha_x \multimap \ell \otimes \alpha'_x)) \quad \multimap \right.$$
$$\left. \bigotimes_{q \in t^\bullet} \bigotimes_{x \in W(t,p)} q(\Gamma_x \otimes \alpha'_x \otimes !(\bar{\ell} \otimes \alpha_x \multimap \ell \otimes \alpha'_x)) \right) \tag{2}$$

– *For each transition $t \in T$ with horizontal synchronisation label $\ell \in L$, construct the factor*

$$\bigotimes_{p \in P} ! \left[p(\iota(\ell,t) \otimes \Gamma) \otimes p(\iota(\bar{\ell},t') \otimes \Gamma') \quad \multimap \right.$$
$$\left. p(\iota(\ell,t) \otimes \Gamma) \otimes p(\iota(\bar{\ell},t') \otimes \Gamma') \right] \tag{3}$$

where

$$\iota(\ell,t) := !(\ell \otimes \bigotimes_{p \in {}^\bullet t} p(W_\otimes(p,t)) \multimap \bar{\ell} \otimes \bigotimes_{q \in t^\bullet} q(\overline{W}_\otimes(t,q)))$$

– *Construct for the current (initial) marking $\mathbf{m}$ and all places $p \in P$ and tokens b with $b \in \mathbf{m}(p)$ the formulae $p(\Psi_{\mathbf{DILL_{PN}}}(b))$. Hence, for the complete marking we have*

$$\bigotimes_{\substack{p \in P \\ b \in \mathbf{m}(p)}} p(\Psi_{\mathbf{DILL_{PN}}}(b)), \tag{4}$$

where multiple occurrences of tokens contribute to multiple occurrences of factors in the tensor product and in the special case of b being a black token $\Psi_{\mathbf{DILL_{PN}}}(b) := \bullet$. For some place p marked by a black token, $p(\bullet)$ is usually abbreviated by p.

The canonical formula of the NP-net is unique up to isomorphism (commutativity of the tensor product).

Thus, for example, the NP-net shown in Figure 3 is translated into the canonical formula composed of the following factors:

From (2):

$$R(\Gamma_x \otimes \alpha_x \otimes !(\mathsf{fill} \otimes \alpha_x \multimap \overline{\mathsf{fill}} \otimes \alpha'_x)) \multimap S(\Gamma_x \otimes \alpha'_x \otimes !(\mathsf{fill} \otimes \alpha_x \multimap \overline{\mathsf{fill}} \otimes \alpha'_x))$$

From (4):

$$R(e \otimes !(\mathsf{fill} \otimes e \multimap \overline{\mathsf{fill}} \otimes f) \otimes !(\mathsf{empty} \otimes f \multimap \overline{\mathsf{empty}} \otimes e))$$

(3) and (1) do not contribute to the canonical formula, since there are neither horizontal synchronisation labels nor autonomous transitions in the example net.

In the example above we did not need factors corresponding to vertical synchronisation of element tokens with inner ones, since the net is only a two-level net. NP-nets provide means for communication with inner and outer nets, i.e. with nets on both upper and lower levels, whereas the canonical formula provides means for communication only with lower nets. Since there are always two nets involved in a synchronisation, both views are equivalent.

Theorem 2. *The sequent*

$$\Psi_{\mathbf{DILL_{PN}}}(\langle \mathcal{N}, \mathbf{m} \rangle) \ \vdash \ \Psi_{\mathbf{DILL_{PN}}}(\langle \mathcal{N}, \mathbf{m}' \rangle),$$

where $\Psi_{\mathbf{DILL_{PN}}}(\langle \mathcal{N}, \mathbf{m} \rangle)$ *is the canonical formula for a marked NP-net* $\langle \mathcal{N}, \mathbf{m} \rangle$*, is derivable in* $\mathbf{DILL}_{\mathrm{PN}}$ *iff* $\mathbf{m}'$ *is reachable from* $\langle \mathcal{N}, \mathbf{m} \rangle$*.*

Proof (sketch). For a canonical formula of an NP-net, synchronisation labels can occur only within the scope of some !-modality. The same holds for implications. For this reason we can disregard all derivations where instances α of some subformula $!\alpha$ from the canonical representation occur in either the antecedent or the succedent of any sequent.

For the remaining sequents it is obvious from the encoding – if rather tedious to show explicitly – that the sequent from Theorem 2 is provable if the respective markings are in the reachability relation for NP-nets. The converse is also true by an argument using the remark given above. □

7 Conclusion

The main focus of our work on object-oriented extensions of Petri nets has been to establish core formalisms that have a clear mathematical semantics and preserve some important decidability results (see [7,8]) that are lost by many *ad hoc* extensions of the basic Petri net formalism. The independent definition of the two similar formalisms of Linear Logic Petri nets and nested Petri nets suggest that the foundations of these formalisms are sound and have a strong theoretical background.

On the other hand, extending Linear Logic to deal with multi-level nested Petri nets leads to the logical framework, which has a natural interpretation not only for Petri nets, but for many applications, where a possibility of using these or other resources depends on their owners and/or locations.

References

1. Farwer B. *A Linear Logic View of Object Petri Nets.* Fundamenta Informaticae, 37:225–246, 1999.
2. Farwer B. *A Multi-Region Linear Logic Based Calculus for Dynamic Petri Net Structure.* Fundamenta Informaticae, 43:61–79, 2000.

3. Farwer B. *Relating Formalisms for Non-Object-Oriented Object Petri Nets.* In Proceedings of the Concurrency Specification and Programming (CS&P'2000) Workshop. 9–11 October 2000. Vol. 1, pp. 53–64. Informatik-Bericht Nr.140, Humboldt-Universität zu Berlin, Informatik-Berichte, Berlin, 2000.
4. Farwer B. *Linear Logic Based Calculi for Object Petri Nets.* Dissertation, Universität Hamburg. Published by: Logos Verlag, ISBN 3-89722-539-5, Berlin, 2000.
5. Girard J.-Y. *Linear Logic.* Theoretical Computer Science, 50:1–102, 1987.
6. Lakos C. A. *From Coloured Petri Nets to Object Petri Nets.* Proc. Int. Conf. on Appl. and Theory of Petri Nets, pp. 278–297. LNCS 935. Springer-Verlag, 1995.
7. Lomazova I. A. *Nested Petri Nets — A Formalism for Specification and Verification of Multi-Agent Distributed Systems.* Fundamenta Informaticae, 43:195–214, 2000.
8. Lomazova I. A. *Nested Petri Nets: Multi Level and Recursive Systems.* In Proceedings of the Concurrency Specification and Programming (CS&P'2000) Workshop. 9–11 October 2000. Vol. 1., pp. 117–128. Informatik-Bericht Nr.140, Humboldt-Universität zu Berlin, Informatik-Berichte, Berlin, 2000.
9. Lomazova I. A. and Schnoebelen Ph. *Some Decidability Results for Nested Petri Nets.* In: Proc. Andrei Ershov 3rd Int. Conf. Perspectives of System Informatics (PSI'99), Novosibirsk, Russia, July 1999, pp. 207–219. LNCS 1755, Springer-Verlag, 1999.
10. Valk R. *Petri Nets as Token Objects: An Introduction to Elementary Object Nets.* In: Proc. Int. Conf. on Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, pp. 1–25, 1998.
11. Zapf M., Heinzl A. *Techniques for Integrating Petri Nets and Object-Oriented Concepts.* Working Papers in Information Systems, No.1/1998. University of Bayreuth, 1998.

Unfoldings of Coloured Petri Nets

Vitaly E. Kozura

A.P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Lavrentjev ave., 630090, Novosibirsk, Russia
vkozura@iis.nsk.su

Abstract. In this paper the unfolding technique is applied to coloured Petri nets (CPN) [6,7]. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven. A new approach consisting in combining unfolding technique with symmetry and equivalence specifications [7] is presented and the important properties of obtained unfoldings are proven. We require CPN to be finite, n-safe and containing only finite sets of colours.

1 Introduction

The state space exploring in Petri net (PN) analysis is one of the most important approaches. Unfortunately, it faces the state explosion problem. Among the approaches which are used to avoid this problem are the stubborn set method, symbolic binary decision diagrams (BDD), methods based on partial orders, methods using symmetry and equivalence properties of the state space, etc. [14].

In [12] McMillan has proposed an unfolding technique for PN analysis. In his works, instead of the reachability graph, a finite prefix of maximal branching process, large enough to describe a system, has been considered. The size of unfolding is exponential in the general case and there are few works which improve in some way the unfolding definitions and the algorithms of unfolding construction [5,8].

Initially McMillan has proposed his method for the reachability and deadlock analysis (which has also been improved in the later work [11]). J.Esparza has proposed a model-checking approach to unfolding of 1-safe systems analysis [3]. In [1] the unfolding technique has been applied to timed PN. In [2,4] LTL-based model-checking on PN's unfolding has been developed. Unfolding of coloured Petri nets has been considered in the general case in [13] for using it in the dependence analysis needed by the Stubborn Set method.

In the present paper the application of the unfolding method based on later works for ordinary PNs to coloured Petri nets (CPN) [6,7] is given. This allows to construct the finite unfolding for CPN and apply the reachability and deadlock analysis methods for it. It is allows to consider applying the model-checking technique to unfoldings of CPN, as well. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven.

In [7] symmetry and equivalence specifications for CPN are introduced. In the present paper it is also presented a new approach consisting in combining unfolding technique with symmetry and equivalence specifications. This allows to reduce additionally the size of CPN's unfolding.

2 Coloured Petri Nets

In this section we briefly remind the basic definitions related to coloured Petri nets and describe the subclass of colours we will use in the paper. More detailed description of CPN can be found in [6].

A *multi-set* is a function m: $S \rightarrow N$, where S is a usual set and N is the set of natural numbers. In the natural way we can define operations such as $m_1 + m_2$, $n{\cdot}m$, $m_1 - m_2$, and relations $m_1 \leq m_2$, $m_1 < m_2$. Also $|m|$ can be defined as $|m| = \sum_{s \in S} m(s)$. Let $Var(E)$ define the set of variables of the expression E, and $Type(E)$ define the type of the expression E.

A *coloured Petri net (CPN)* is the net $\mathbf{N} = (S, P, T, A, N, C, G, E, I)$, where S, P, T, A are the sets of colours, places, transitions, and arcs such that $P \cap T = P \cap A = T \cap A = \emptyset$, N is a mapping $N : A \rightarrow (P \times T) \cup (T \times P)$, C is a colour function $C : P \rightarrow S$, G is a guard function such that for all $t \in T$ $Type(G(t)) = bool$ and $Type(Var(G(t))) \subseteq S$, E is the function defined on arcs with $Type(E(a)) = C(p)_{MS}$, where p is the place from $N(a)$ and $Type(Var(E(a))) \subseteq S$ and I is the initial function defined on places, such that for all $p \in P$ $Type(I(p)) = C(p)_{MS}$.

$A(t), Var(t), A(x, y), E(x, y)$ can be defined in the natural way.

A *binding* b is a function from $Var(t)$ such that $b(v) \in Type(v)$ and $G(t)\langle b \rangle$. The set of bindings for t will be denoted by $B(t)$. A *token element* is a pair (p, c) where $p \in P$ and $c \in C(p)$. The set of all token elements is denoted by TE. A *binding element* is a pair (t, b) where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE. A *marking* M is a multi-set over TE. A *step* Y is a multi-set over BE. A step Y is *enabled* in the marking M if for all $p \in P$ $\sum_{(t,b) \in Y} E(p, t)\langle b \rangle \leq M(p)$ and a new marking M_1 is given by $M_1(p) = M(p) - \sum_{(t,b) \in Y} E(p, t)\langle b \rangle + \sum_{(t,b) \in Y} E(t, p)\langle b \rangle$.

Now we can define a subclass of coloured Petri nets, which is large enough to describe many interesting systems and still allows us to build a finite prefix of its branching process. The detailed description can be found in [9]. The set of basic colour domains is obtained from the types of *Standard ML (SML)* [6] by allowing to consider only finite colour domains $s \in S$. All functions defined in [6] and having the above described classes as their domains are allowed in our subclass. The CPN satisfying all the above-mentioned requirements is called *S-finite*.

The marking M of a CPN is *n-safe* if $|M(p)| \leq n$ for all $p \in P$. A CPN is called *n-safe* if all of its reachable markings are n-safe. 1-safe net is also called *safe*. A *preset* of an element $x \in P \cup T$ denoted by ${}^\bullet x$ is the set ${}^\bullet x = \{y \in P \cup T \mid \exists a :$

$N(a) = (y, x)\}$. A *postset* of x denoted by $x^\bullet$ is the set $x^\bullet = \{y \in P \cup T \mid \exists a : N(a) = (x, y)\}$.

The CPN considered in this paper are the CPN satisfying three additional properties:

1. The number of places and transitions is finite.
2. The CPN is n-safe.
3. The CPN is S-finite.

3 Branching Process of Coloured Petri Nets

Let $\mathbf{N}$ be a Petri net. We will use the term *nodes* for both places and transitions. The nodes x_1 and x_2 are *in conflict*, denoted by $x_1 \sharp x_2$, if there exist transitions t_1 and t_2 such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and (t_1, x_1) and (t_2, x_2) belong to the transitive closure of N (which we denote by R_t). The node x is in *self-conflict* if $x \sharp x$. We will write $x_1 \leq x_2$ if $(x_1, x_2) \in R_t$ and $x_1 < x_2$ if $x_1 \leq x_2$ and $x_1 \neq x_2$. We say that x *co* y, or $x \| y$, or x *concurrent* y if neither $x < y$ nor $x > y$ nor $x \sharp y$.

An Occurrence Petri Net (OPN) is an ordinary Petri net $\mathbf{N} = (P, T, N)$, where

1. P, T are the sets of places and transitions,
2. $N \subseteq (P \times T) \cup (T \times P)$ gives us the incidence function,

satisfying the following properties:

1. For all $p \in P$ $|p| \leq 1$,
2. N is acyclic, i.e., the (irreflexive) transitive closure of N is a partial order.
3. N is finitely preceded, i.e. for all $x \in P \cup T$ the set $\{y \in P \cup T \mid y \leq x\}$ is finite which gives us the existence of $Min(\mathbf{N})$, the set of minimal elements of $\mathbf{N}$ with respect to R_t.
4. no transition is in self conflict.

Let $\mathbf{N}_1 = (P_1, T_1, N_1)$ and $\mathbf{N}_2 = (P_2, T_2, N_2)$ be two Petri nets. A *homomorphism* h from $\mathbf{N}_2$ to $\mathbf{N}_1$ is a mapping $h : P_2 \cup T_2 \to P_1 \cup T_1$ such that

1. $h(P_2) \subseteq P_1$ and $h(T_2) \subseteq T_1$.
2. for all $t \in T_2$ $h|_{{}^\bullet t} = {}^\bullet t \to {}^\bullet h(t)$.
   for all $t \in T_2$ $h|_{t^\bullet} = t^\bullet \to h(t)^\bullet$.

Now we give the main definition of the section. This is the first novelty of the paper, a formal definition of a branching process for coloured Petri nets. After the following definition, the existence result is given.

Definition 1 *A branching process of a CPN $\mathbf{N}_1 = (S_1, P_1, T_1, A_1, N_1, C_1, G_1, E_1, I_1)$ is a tuple $(\mathbf{N}_2, h, \varphi, \eta)$, where $\mathbf{N}_2 = (P_2, T_2, N_2)$ is an OPN, h is a homomorphism from $\mathbf{N}_2$ to $\mathbf{N}_1$, φ and η are the functions from P_2 and T_2, respectively, such that*

1. $\varphi(p){\in}C_1(h(p))$.
2. $\eta(t){\in}B(h(t))$.
Other requirements are listed below:
3. *for all* $p_1{\in}P_1$ $\sum_{p\in Min(\mathbf{N}_2)\ \mid\ h(p)=p_1}\varphi(p) = M_0(p_1)$,
4. $G_1(h(t))\langle\eta(t)\rangle$ *for all* $t{\in}T_2$.
5. $\forall t'{\in}T_2 \mid (\exists a{\in}A_1 : N_1(a) = (p,t)$ *and* $h(t') = t) \Longrightarrow$
$E_1(a)\langle\eta(t')\rangle = \sum_{(p'\in{}^\bullet t'\ \mid\ h(p')=p)}\varphi(p')$,
$\forall t'{\in}T_2 \mid (\exists a{\in}A_1 : N_1(a) = (t,p)$ *and* $h(t') = t) \Longrightarrow$
$E_1(a)\langle\eta(t')\rangle = \sum_{(p'\in t'^\bullet\ \mid\ h(p')=p)}\varphi(p')$,
6. *If* $(h(t_1) = h(t_2))$ *and* $(\eta(t_1) = \eta(t_2))$ *and* $({}^\bullet t_1 = {}^\bullet t_2)$ *then* $t_1 = t_2$.

Using the first two properties, we can associate a token element (p,c) of $\mathbf{N}_1$ with every place in $\mathbf{N}_2$ and the binding element (t,b) of $\mathbf{N}_1$ with every transition in $\mathbf{N}_2$. So we can further consider the net $\mathbf{N}_2$ as containing the places which we identify with token elements of $\mathbf{N}_1$, and transitions which we identify with binding elements of $\mathbf{N}_1$. So we sometimes use them instead, like $h((t,b)) = t$ means $h(t') = t$ and $\eta(t') = b$ or $p{\in}{}^\bullet(t,b)$ means $p{\in}{}^\bullet t'$ and $h(t') = t$ and $\eta(t') = b$. Analogously, we can consider $(p,c){\in}P_2$ as $p'{\in}P_2$ and $h(p') = p$ and $\varphi(p) = c$. Also, $h(p,c) = p$ and $h(t,b) = t$.

It can be shown that any finite CPN has a maximal branching process (MBP) up to isomorphism (theorem 1). We can declare existence of the maximal branching process when considering the algorithm of its generation. The algorithm is described in [9] and the following theorem is proven there.

Theorem 1 *For a given CPN* $\mathbf{N}$ *there exists a maximal branching process MBP(*$\mathbf{N}$*)*

This branching process can be infinite even for the finite nets if they are not acyclic. We are interested in finding a finite prefix of a branching process large enough to represent all the reachable markings of the initial CPN. This finite prefix will be called an unfolding of the initial CPN.

4 Unfoldings of CPN

A *configuration* C of an OPN $\mathbf{N} = (P,T,N)$ is a set of transitions such that $t{\in}C$ $\Longrightarrow$ for all $t_0{\le}t$, where $t_0{\in}C$ and for all $t_1,t_2{\in}C$ $\neg(t_1\sharp t_2)$. A set $X_0{\subseteq}X$ of nodes is called a *co-set*, if for all $t_1,t_2{\in}X_0$: (t_1 co t_2). A set $X_0{\subseteq}X$ of nodes is called a *cut*, if it is a maximal co-set with respect to the set inclusion.

Finite configurations and cuts are closely related. Let C be a finite configuration of an occurrence net, then $Cut(C) = (Min(\mathbf{N}){\cup}C^\bullet)^\bullet C$ is a cut.

Let $\mathbf{N}_1 = (S_1,P_1,T_1,A_1,N_1,C_1,G_1,E_1,I_1)$ be a CPN and MBP($\mathbf{N}_1$) = $(\mathbf{N}_2,h,\varphi,\eta)$, where $\mathbf{N}_2 = (P_2,T_2,N_2)$, be its maximal branching process. Let C be a configuration of $\mathbf{N}_2$. We define a marking *Mark(C)* which is a marking of $\mathbf{N}_1$ such that $Mark(C)(p) = \sum_{(p'\in Cut(C)\ \mid\ h(p')=p)}M_2(p')$.

Let $\mathbf{N}$ be an OPN. For all $t{\in}T$ the configuration $[t] = \{t'{\in}T \mid t'{\le}t\}$ is called a *local configuration.* (The fact that [t] is a configuration can be easily checked).

Let us consider the maximal branching process for a given CPN. It can be noticed that MBP($\mathbf{N}$) satisfies the completeness property, i.e., for every reachable marking M of $\mathbf{N}$ there exists a configuration C of MBP($\mathbf{N}$) (i.e., C is the configuration of OPN) such that $Mark(C) = M$. Otherwise we could add a necessary path and generate a larger branching process. This would be a contradiction with the maximality of MBP($\mathbf{N}$).

Now we are ready to define three types of cutoffs used in the definition of unfolding. The first two definitions for ordinary PNs can be found in [3,12]. The last is the definition given in [8].

Definition 2 *Let $\mathbf{N}$ be a coloured Petri net and MBP($\mathbf{N}$) be its maximal branching process. Then*

1. *A transition $t \in T$ of an OPN is a GT_0-cutoff, if there exists $t_0 \in T$ such that $Mark([t]) = Mark([t_0])$ and $[t_0] \subset [t]$.*
2. *A transition $t \in T$ of an OPN is a GT-cutoff, if there exists $t_0 \in T$ such that $Mark([t]) = Mark([t_0])$ and $||[t_0]|| < ||[t]||$.*
3. *A transition $t \in T$ of an OPN is a EQ-cutoff, if there exists $t_0 \in T$ such that*
   a) $Mark([t]) = Mark([t_0])$
   b) $||[t_0]|| = ||[t]||$
   c) $\neg(t \| t_0)$
   d) there are no EQ-cutoffs among t' such that $t' \| t_0$ and $||[t']|| \leq ||[t_0]||$.

Definition 3 *For a coloured Petri net $\mathbf{N}$, an unfolding is obtained from the maximal branching process by removing all the transitions t', such that there exists a cutoff t and $t < t'$, and all the places $p \in t'^{\bullet}$. If Cutoff = GT_0(GT)-cutoffs, then the resulted unfolding is called GT_0(GT)-unfolding. GT_0(GT)-unfolding is also called the McMillan unfolding. If Cutoff = GT-cutoffs $\cup$ EQ-cutoff, then the resulted unfolding is called EQ-unfolding.*

It has been shown that the McMillan unfoldings are inefficient in some cases. The resulting finite prefix grows exponentially, when the minimal finite prefix has only a linear growth. The following proposition can be formulated for these three types of unfoldings ([9]).

Proposition 1 *EQ-unfolding $\leq$ GT-unfolding $\leq$ GT_0-unfolding.*

The following theorem presents the main result of this section ([9]).

Theorem 2 *Let $\mathbf{N}$ be a CPN. Then for its unfoldings we have:*

1. *EQ-unfolding, GT-unfolding and GT_0-unfolding are finite.*
2. *EQ-unfolding, GT-unfolding and GT_0-unfolding are safe, i.e., if C and C' are configurations, then $C \subseteq C' \implies Mark(C') \in [Mark(C)\rangle$.*
3. *EQ-unfolding, GT-unfolding and GT_0-unfolding are complete, i.e., $M \in [M_0\rangle \implies$ there exists a configuration C such that Mark(C) = M.*

In the general case the algorithm proposed in [12] and applied to coloured Petri nets in [9] has an exponential complexity. The algorithm from [8] is rather efficient in the speed of unfolding generation. In the case of an ordinary PN it gives the overall complexity $O(N_P{\cdot}N_T)$, where N_P and N_T are the numbers of places and transitions in EQ-unfolding. This algorithm was also transferred to coloured Petri nets [9] and a close estimation holds if we don't take into consideration the calculation complexity of arc and guard functions. In this case we obtain $O(N_P{\cdot}N_T{\cdot}B)$, where $B = max\{|B(t)| \ : \ t{\in}T_{CPN}\}$.

5 Unfoldings with Symmetry and Equivalence

In this part the technique of equivalence and symmetry specifications for coloured Petri nets (CPN) will be applied to the unfolding nets of CPN. It will be shown how to generate the maximal branching process and its finite prefixes for a given CPN under the equivalence or symmetry specifications. All symmetry and equivalence specifications are taken from [6].

Let $\mathbf{N}$ be a CPN and $\mathbf{M}$ and BE be the sets of all markings and binding elements of $\mathbf{N}$. The pair $(\approx_M, \approx_{BE})$ is called an *equivalence specification* if $\approx_M$ is an equivalence on $\mathbf{M}$ and $\approx_{BE}$ is an equivalence on BE. $M_\approx$ and $BE_\approx$ are the equivalence classes. We say $(b, M){\approx}(b^*, M^*)$ iff $b{\approx_{BE}}b^*$ and $M{\approx_M}M^*$. Let us have $X{\subseteq}\mathbf{M}$ and $Y{\subseteq}M_\approx$, then we can define: [X] = $\{M{\in}\mathbf{M} \mid \exists x{\in}X : M{\approx_M}x\}$ — the set of all markings equivalent to the markings from X and [Y] = $\{M{\in}\mathbf{M} \mid \exists y{\in}Y : M{\in}y\}$ — the set of all markings from the classes from Y. The equivalence specification is called *consistent* if for all $M_1, M_2{\in}[[M_0\rangle]$ we have $M_1{\approx_M}M_2 \implies [Next(M_1)] = [Next(M_2)]$, where $Next(M_1) = \{(b, M){\in}BE{\times}\mathbf{M}| \ M_1[b\rangle M\}$.

A *symmetry specification* for a CP-net is a set of functions $\Phi \subseteq [\mathbf{M}{\cup}BE \to \mathbf{M}{\cup}BE]$ such that $(\Phi, \bullet)$ is an algebraic group and $\forall\phi{\in}\Phi : \phi|_{\mathbf{M}}{\in}[\mathbf{M}{\to}\mathbf{M}]$ and $\phi|_{BE}{\in}[BE{\to}BE]$. Each element of Φ is called a *symmetry*. A symmetry specification Φ is it consistent iff the following properties are satisfied for all symmetries $\phi{\in}\Phi$, all markings $M_1, M_2{\in}[M_0\rangle$ and all binding elements $b{\in}BE$ $\phi(M_0) = M_0$ and $M_1[b\rangle M_2{\Longleftrightarrow}\phi(M_1)[\phi(b)\rangle\phi(M_2)$.

Now the cutoff criteria will be defined for a CPN with a symmetry specification Φ or equivalence specification ϕ. We call the finite prefix of the maximal branching process of CPN obtained by using new cutoff criteria an *unfolding with symmetry* Unf^{Φ} or *unfolding with equivalence* $Unf^{\approx}$. Since accordingly to described above we can consider the symmetry specification as the case of equivalence specifications, we give the cutoff definitions only for equivalence specifications.

Taking into consideration the consistency of the regarded equivalence, we can conclude that it is sufficient to consider the classes [M] in our definitions of cutoffs. The classes of binding elements will be obtained in a natural way.

Definition 4 *Let* $\mathbf{N}$ *be a coloured Petri net and MBP(*$\mathbf{N}$*) be its maximal branching process. Then*

1. *A transition $t{\in}T$ of an OPN is a $GT_0{}^{\approx}$-cuttoff if there exists $t_0{\in}T$ such that $Mark([t]){\approx}Mark([t_0])$ and $[t_0]{\subset}[t]$.*
2. *A transition $t{\in}T$ of an OPN is a $GT^{\approx}$-cutoff if there exists $t_0{\in}T$ such that $Mark([t]){\approx}Mark([t_0])$ and $|[t_0]| < |[t]|$.*
3. *A transition $t{\in}T$ of an OPN is a $EQ^{\approx}$- cutoff if there exists $t_0{\in}T$ such that*
   a) $Mark([t]){\approx}Mark([t_0])$
   b) $|[t_0]| = |[t]|$
   c) $\neg(t \| t_0)$
   d) there are no EQ-cutoffs among t' such that $t' \| t_0$ and $|[t']|{\leq}|[t_0]|$.

The notion $Unf^{\approx}$ is used for any type of unfoldings.

Proposition 2 *$EQ^{\approx}$-unfolding $\leq$ $GT^{\approx}$-unfolding $\leq$ $GT_0{}^{\approx}$-unfolding.*

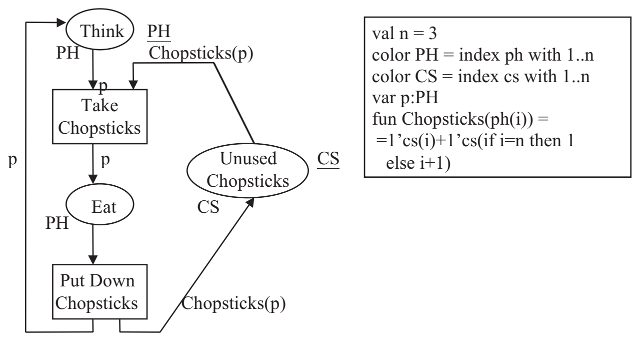
The following theorem presents the main result of this section [9].

Theorem 3 *Let $\mathbf{N}$ be a CPN and $\approx = (\approx_M, \approx_{BE})$ be a consistent equivalence on $\mathbf{N}$. Then for an $Unf^{\approx}(\mathbf{N})$ we have:*

1. *$[M]{\in}[[M_0]\rangle \iff \exists C$, a configuration of $Unf^{\approx}(\mathbf{N})$ | $Mark(C){\approx}_M M$.*
2. *$C{\subset}C'$ and C' is a configuration of $Unf^{\approx}(\mathbf{N}) \iff [Mark(C')]{\in}[[Mark(C)]\rangle$.*

6 Net Examples

Figures 1, 2 show us the dining philosophers CPN and its unfolding with the symmetry specification.



```
val n = 3
color PH = index ph with 1..n
color CS = index cs with 1..n
var p:PH
fun Chopsticks(ph(i)) =
 =1'cs(i)+1'cs(if i=n then 1
   else i+1)
```

Fig. 1. The dining philosophers example

As is seen from the pictures, using the symmetry we obtain a much smaller prefix when constructing the $Unf^{\approx}$. However the size of the state space of the respective OE-graph (O-graph with equivalence) for this example is just two states.

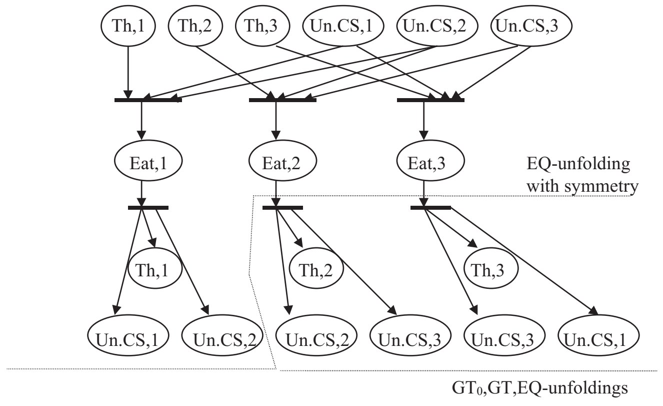


Fig. 2. The unfolding with symmetry

To avoid an impression that given an equivalence specification it is inefficient to use unfoldings let us consider the CPN representing the producer-consumer system [7] (see Appendix). We consider the case when the buffer capacity $nb = 1$. As an equivalence specification, the abstraction from the data d_1 and d_2 is considered. The table 1 represents the results.

Let us notice that we should generate EQ-unfolding when using the symmetry specification. In the case of equivalence specifications in general we can use all types of unfoldings.

7 Conclusion

In this paper the unfolding technique proposed by McMillan in [12] and developed in later works is applied to coloured Petri nets as they are described in [6,7]. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven. All the necessary details are presented in [9].

The unfolding is a finite prefix of the maximal branching process. To truncate the occurrence net, we consider three cutoff criteria in the paper. To construct the finite prefix, two algorithms of unfolding generation were formally transferred from the ordinary PN's area [9]. The complexities of these algorithms are discussed in this paper.

One of the important novelties of the paper is the application of the unfolding technique to CPN with symmetry and equivalence specifications as they are represented in [7]. The size of unfolding is often much smaller than the size of the reachability graph of a PN. Using the symmetry and equivalence specifications in the unfolding generation, we can additionally reduce the size of CPN's unfolding.

We require a CPN to be finite, n-safe and to contain only finite sets of colours.

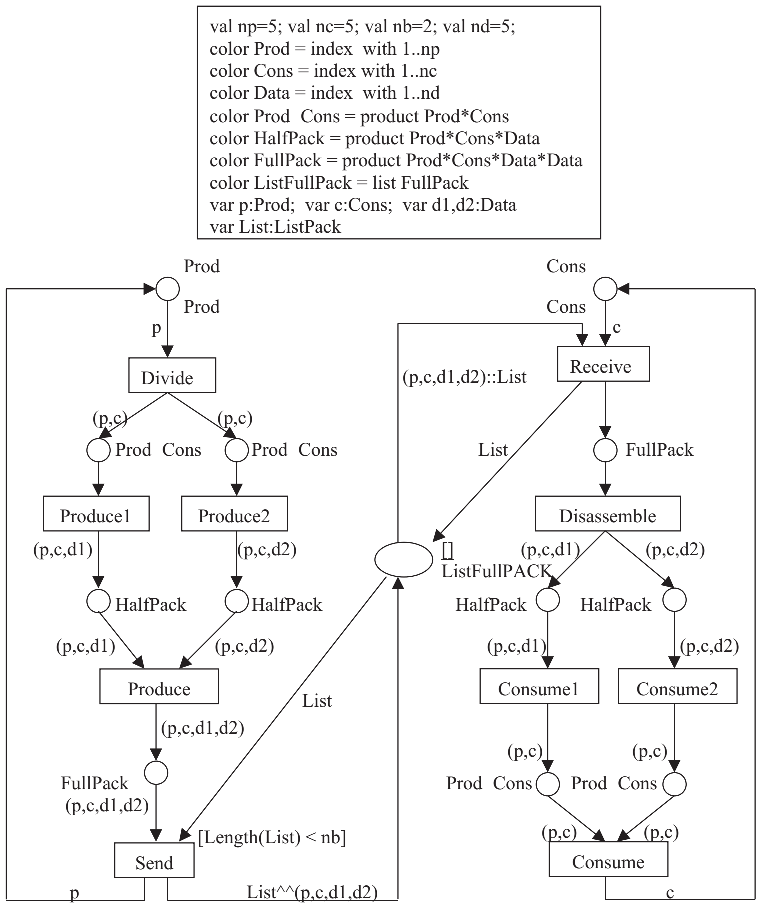
In the future it is planned to construct finite unfoldings of Timed CPN as they are described in [6], using the technique of unfolding with equivalence (the first results are already obtained in [10]), and also to make all the necessary experiments with unfoldings of coloured Petri nets including the implementation of the model-checking method.

Acknowledgments. I would like to thank Valery Nepomniaschy for drawing my attention to this problem and Elena Bozhenkova for valuable remarks.

References

1. **B.Bieber, H.Fleischhack** Model Checking of Time Petri Nets Based on Partial Order Semantics. Proc. CONCUR'99. — Berlin a.o.: Springer-Verlag, 210-225 (1999).
2. **J.-M.Couvreur, S.Grivet, D.Poitrenaud:** Designing an LTL Model-Checker Based on Unfolding Graphs. Lecture Notes in Computer Science, Vol.1825, 123-145 (2000).
3. **J.Esparza:** Model-Checking Using Net Unfoldings. Lecture Notes in Computer Science Vol.668, 613-628 (1993).
4. **J.Esparza J, K.Heljanko:** A New Unfolding Approach to LTL Model-Checking. Lecture Notes in Computer Science Vol.1853, 475-486 (2000).
5. **J.Esparza, S.Romer, W.Vogler:** An Improvement of McMillan's Unfolding Algorithm Proc. TACAS'96. — Berlin a.o.: Springer-Verlag, pp. 87-106 (1997).
6. **K.Jensen:** Coloured Petri Nets. Vol. 1. Berlin a.o.: Springer, (1995).
7. **K.Jensen:** Coloured Petri Nets. Vol. 2. Berlin a.o.: Springer, (1995).
8. **A.Kondratyev, M.Kishinevsky, A.Taubin, S.Ten:** A Structural Approach for the Analysis of Petri Nets by Reduced Unfoldings. 17th Intern. Conf. on Application and Theory of Petri Nets, Osaka, June 1996. Berlin a.o.: Springer-Verlag, 346-365 (1996).
9. **V.E.Kozura:** Unfoldings of Coloured Petri Nets. Tech. Rep. N 80, Novosibirsk 34 pages (2000).
10. **V.E.Kozura:** Unfoldings of Timed Coloured Petri Nets. Tech. Rep. N 82, Novosibirsk 33 pages (2001).
11. **S.Melzer, S.Romer:** Deadlock Checking Using Net Unfoldings. Proc. of the Conf. on Computer Aided Verification (CAV'97), Haifa, pp. 352-363 (1997).
12. **K.L.McMillan:** Using Unfolding to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. Lecture Notes in Computer Science Vol.663, 164-174 (1992).
13. **A.Valmari:** Stubborn Sets of Coloured Petri Nets. Proc. of the 12th Intern. Conf. On Application and Theory of Petri Nets. — Gjern, 102-121 (1991).
14. **A.Valmari:** The State Explosion Problem. Lecture Notes in Computer Science Vol.1491, 429-528 (1998).

Appendix



```
val np=5; val nc=5; val nb=2; val nd=5;
color Prod = index  with 1..np
color Cons = index with 1..nc
color Data = index  with 1..nd
color Prod  Cons = product Prod*Cons
color HalfPack = product Prod*Cons*Data
color FullPack = product Prod*Cons*Data*Data
color ListFullPack = list FullPack
var p:Prod;  var c:Cons;  var d1,d2:Data
var List:ListPack
```

Fig. 3. Producer-Consumer system

This net example is taken from [7] and represents the producer-consumer system. We consider the case when nb=1. Let us calculate here the sizes of reachability graphs and unfoldings for the producer-consumer system.

The number of reachable markings is
$N = (1 + nc + 2{\cdot}nc{\cdot}nd + 2{\cdot}nc{\cdot}nd^2)^{np}{\cdot}(1 + np + 2{\cdot}np{\cdot}nd + 2{\cdot}np{\cdot}nd^2)^{nc}{\cdot}(1 + np{\cdot}nc{\cdot}nd^2)$.

The unfolding consists of four parts (we denote it by PA, PB, PC and CA). When a producer initially produces data, the part PA is working. Part PB may work after a producer laid the first data to the buffer, but a consumer still cannot begin his part. Finally, PC is the part when a consumer definitely begins his work and a producer fulfills the buffer again. A consumer has the unique part CA. We have $|PA| = |PB| = |CA| = 5$ and $|PC| = 4$. The whole size is 19. When adding either one more producer or one more consumer, we come to the situation of doubling of $|PA|$, $|PB-1|$ and $|CA|$ and adding the square of the number of parts $|PC+1|$. Adding one more data acts as adding the square number of possibilities. Finally the size of the unfolding is

$$UnfSize = |PA|{\cdot}np{\cdot}nc{\cdot}nd^2 + |CA|{\cdot}np{\cdot}nc{\cdot}nd^2 + |PB-1|{\cdot}np{\cdot}nc{\cdot}nd^2 + |PC+1|{\cdot}(np{\cdot}nc{\cdot}nd^2)^2.$$

As an equivalence specification, the abstraction from the data d_1 and d_2 is considered. For a graph this means that we can put nd=1. In the case of unfolding we obtain the additional (nd-1) transitions in the part PA. The whole size of EQ-unfolding with this equivalence is

$$UnfSize^{\approx} = |PA|{\cdot}np{\cdot}nc + |CA|{\cdot}np{\cdot}nc + |PB-1|{\cdot}np{\cdot}nc + |PC+1|{\cdot}(np{\cdot}nc)^2 + (nd-1).$$

The table below represents the results.

Table 1.

np	nc	nd	O-graph	Consistent OE-graph	Unfolding's Size	EQ-unfolding with equivalence
1	1	1	72	72	19	19
3	3	3	$1.58{\cdot}10^{13}$	$1.68{\cdot}10^{8}$	$3.3{\cdot}10^{4}$	533
10	10	5	$1.32{\cdot}10^{59}$	$1.43{\cdot}10^{36}$	$3.1{\cdot}10^{7}$	$5.14{\cdot}10^{4}$
10	10	10	$7.8{\cdot}10^{70}$	$1.43{\cdot}10^{36}$	$5.0{\cdot}10^{8}$	$5.14{\cdot}10^{4}$
20	20	20	$1.73{\cdot}10^{174}$	$5.97{\cdot}10^{82}$	$1.28{\cdot}10^{11}$	$8.0{\cdot}10^{5}$
50	50	20	$2.11{\cdot}10^{469}$	$2.32{\cdot}10^{243}$	$5.0{\cdot}10^{12}$	$3.1{\cdot}10^{7}$

A Net-Based Multi-tier Behavior Inheritance Modelling Method*

Shengyuan Wang, Jian Yu, and Chongyi Yuan

Department of Computer Science and Technology,
Peking University, Beijing, 100871, China
`Wwssyy@263.net, yuj@theory.cs.pku.edu.cn, Lwyuan@pku.edu.cn`

Abstract. A multi-tier methodology is required in order to make a smooth transformation from one stage to the next in the course of software development under a consistent conceptual framework. We present, in this paper, a multi-tier behavior inheritance modelling method based on Petri Nets, which is illustrated through STLEN and DCOPN that are two net models serving as the tools for describing behaviors at two consecutive modelling tiers respectively.

Keywords: Concurrency, Object Orientation, Petri Net, Behavior Inheritance, Modelling Method.

1 Introduction

The integration of Petri Nets with object orientation techniques has become promising ([1]–[8]). Parallelism, concurrency and synchronization are easy to model in terms of Petri Nets, and many techniques and software tools are already available for Petri Net analysis. These advantages have made Petri nets quite suitable to model the dynamic behavior of concurrent objects.

A concurrent object-oriented system consists of a dynamically varying configuration of concurrent objects operating in parallel. For different stages in the software development of such sort of systems, diversified Petri Net models may be found in the literature to perform the specification and/or verification of system behavior respecting that stage. Some net models are suitable to be used during the earlier stages ([5][7][8]), and others during later ones ([1][3][4][6]). Up to now, however, there is a lack of net-based formal methods that will guarantee a smooth transformation from one stage to the next under a consistent conceptual framework. This has prevented net-based methods from being more widely applied to the modelling of a concurrent object-oriented system, since incremental developing is one of the principles in object-oriented methodology. So we are persistent on the opinion that multi-tier methodology is necessary. Each tier is a significant specification phase in which one net-based model may be chosen as the modelling language. A multi-tier method should guarantee that the system

* This work was Supported by the National Natural Science Foundation of China under grant No. 69973003, and by the China NKBRSF (973) under grant G1999032706.

behavior specified in one tier to be preserved in its successor tier, though the specification in the successor tier is usually more detailed.

A practical multi-tier method should take into account all the primary elements of concurrent object-orientation, such as object (class) representation, inheritance, aggregation, cooperation (association), concurrency (intra/inter objects), etc. In this paper, we present a multi-tier behavior inheritance modelling method, and two Petri Net models, belonging to two tiers respectively, serve as the illustration of this method. The net model in the super-tier is a modified EN-system, called STLEN, in which both S-elements and T-elements are labelled. And in the successor tier (or sub-tier) is a net model, called DCOPN , which has s flavor of concurrent object-oriented programming languages, like net models in [1], [3] or [4].

The net models STLEN and DCOPN are briefly introduced in section 2 and section 3 respectively. In section 4, the multi-tier behavior inheritance modelling method is discussed, illustrated by the two-tier method. And finally in section 5, a general engineering guide (steps) for the method is presented as the conclusion.

2 Object Petri Net Model STLEN

STLEN is an extended net model to the Elementary Net System[14] model, with a higher abstraction tier. A STLEN system is obtainable through labelling an EN-system, both the S-elements and T-elements being labelled. The labelling leads to dividing the elements into groups, which makes a STLEN system possess a particular abstraction for methods and attributes, i.e., an observable group of S-elements may compare with a subset of the attribute set, and an observable group of T-elements may be compared to a method.

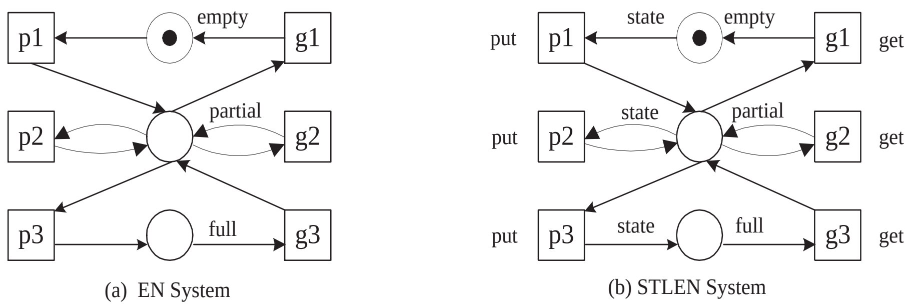


Fig. 2.1. Bounded buffer

Fig 2.1 (a) is an EN system that is the behavior specification of a bounded buffer. The state of a buffer is described with three S-elements *empty*, *partial* and *full* (the buffer size is assumed to be greater than 2). The T-elements $p1$,

$p2$ and $p3$ are used to represent the possible *put* actions in different cases of the buffer state, and the T-elements $g1$, $g2$ and $g3$ are related to *get* actions in the same meaning.

In building an object model for a bounded buffer, it is natural to use an attribute group named *state* to hold the current state of the buffer, and to let it have two public methods *put* and *get*. Fig 2.1(b) is a ST-Labelled EN system whose underlying EN system is the one in fig 2.1(a). In this STLEN system, the T-elements $p1$, $p2$, $p3$ are labelled by *put*, and $g1$, $g2$, $g3$ are labelled by *get*; the S-elements *empty*, *partial*, $full$ are labelled by *state*

Fig 2.2 is another example of STLEN systems. A lockable bounded buffer is a bounded buffer that has one more attribute group about the state of its "*lock*", and two more methods, which can change the state of this attribute. As an example for discussing the inheritance in this paper, the class (templet) of lockable bounded buffers is supposed to inherit the class (templet) of bounded buffers. For the sake of the inheritance depicted in fig 2.2(a), the STLEN system diagram of a lockable bounded buffer in fig 2.2(b) is a reduced version by leaving out the unchanged part inherited from the diagram in fig 2.1(b).

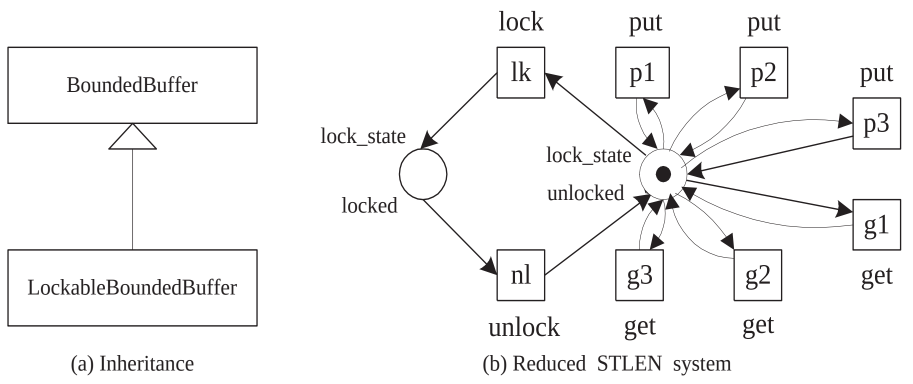
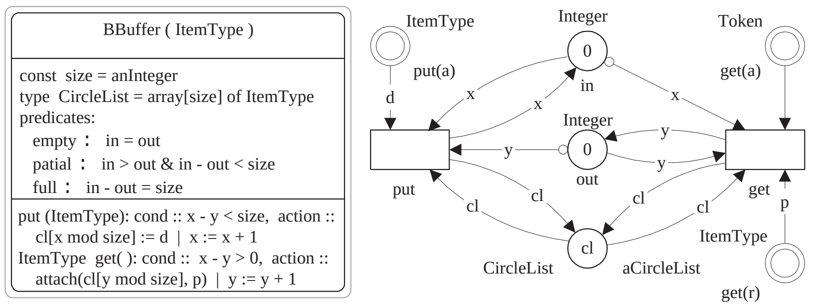


Fig. 2.2. Lockable bounded buffer

3 Object Petri Net Model DCOPN

DCOPN is a colored Petri net model, with a relatively low abstraction tier. By contrast to the STLEN model, DCOPN is enriched by many static details. A DCOPN class net is a net structure specifying the dynamic behavior of a class / object. A DCOPN system consists of the instances of many DCOPN class nets and has a dynamic configuration.

The right part of fig 3.1 is a DCOPN class net diagram for a bounded buffer. Doubled circles are *method port places*. *Put* is a *call* like method, which has one *accept* port place, *put*(*a*). *Get* is a *call*/*return* like method has two method port places, one *result* port place, *get*(*r*), and one *accept* port place, *get*(*a*). Some places serve as *state port places*, which can be used for constructing *attribute predicates*. For example, *in* and *out* are two state port places, which are used in the definitions of the attribute predicate *empty*, *partial* and *full*. An attribute predicate may be accessed (read only) by outside of the object (an instance of the class net). The color (type) name of a place is written on the near top or left of the place. Arcs (starting from a place) with a small circle at their beginning ends are test arcs ([11]).

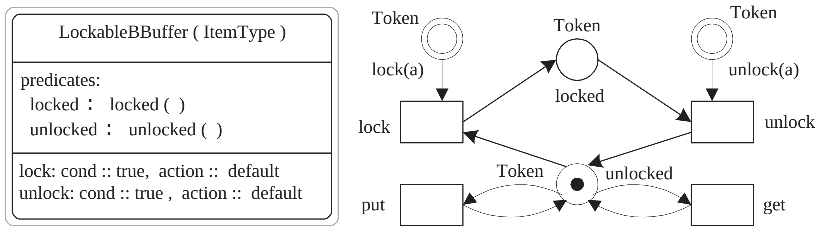

BBuffer (ItemType)

const size = anInteger
type CircleList = array[size] of ItemType
predicates:
empty : in = out
patial : in > out & in - out < size
full : in - out = size

put (ItemType): cond :: x - y < size, action ::
cl[x mod size] := d | x := x + 1
ItemType get( ): cond :: x - y > 0, action ::
attach(cl[y mod size], p) | y := y + 1

Fig. 3.1. DCOPN class net of a bounded buffer

The left part of fig 3.1 includes the declarations for the class net in the right part of the figure, constants, types, attribute predicates, method signatures, guard conditions and action expressions of T-elements, etc. The *attach* function would attach a reference to an instance object. In DCOPN, a token representing an object is valued to a *reference* of that object, instead of its *identifier*.


LockableBBuffer (ItemType)

predicates:
locked : locked ( )
unlocked : unlocked ( )

lock: cond :: true, action :: default
unlock: cond :: true , action :: default

Fig. 3.2. DCOPN class net of locable bounded buffer

Fig 3.2 represents a DCOPN specification for a lockable bounded buffer. Similar to the situation in fig 2.2, fig 3.2 is a reduced version, with both the declaration and the diagram parts being reduced for the sake of the inheritance.

4 Multi-tier Inheritance Modelling Method

4.1 Behavior Preserving Relation between Specification Tiers

Let DCOPN be the subsequent net model of STLEN in our multi-tier modelling method. For the behavior preserving from a STLEN tier net to its corresponding DCOPN one, a *restricted bisimulation relation*, denoted by $\cong$, between them has been defined. The word *restricted* conforms to the incremental design process from the STLEN tier to the DCOPN tier, i.e., the specification in the later is more detailed, or more restricted. Furthermore, the definition of the relation is based on the grouping of S-elements.

Let Σ_1 be the STLEN system in fig 2.1(b), and Σ_1' be the DCOPN class net in fig 3.1. A relation

$$R_{state} = \{\langle\{empty\}, empty\rangle, \langle\{partial\}, partial\rangle, \langle\{full\}, full\rangle\}$$

can serve as a restricted bisimulation, which is a relation from $\wp(\{empty, partial, full\})$, the powerset of all the S-elements in the S-elements group *state* in Σ_1, to $\{empty, partial, full\}$, a subset of all the attribute predicates in Σ_1'. Since *state* is the only S-elements group, we have $\Sigma_1 \cong \Sigma_1'$.

Let Σ_2 be the STLEN system in fig 2.2(b), and Σ_2' be the DCOPN class net in fig 3.2. For the S-elements group *state* and *lock_state* in Σ_2, relations

$$R_{state} = \{\langle\{empty\}, empty\rangle, \langle\{partial\}, partial\rangle, \langle\{full\}, full\rangle\}$$

and

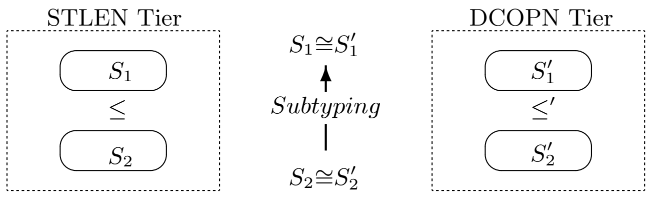
$$R_{lock\_state} = \{\langle\{locked\}, locked\rangle, \langle\{unlocked\}, unlocked\rangle\}$$

can serve as the restricted bisimulations respectively. So we have $\Sigma_2 \cong \Sigma_2'$.

4.2 Subtyping Relation Preserving

The term *inheritance* has often twofold meanings in the literature: the “code” reuse and the behavior preserving. In many places and also in this paper, the later is the meaning for another term *subtyping*, and the term inheritance stands simply for the former. In the situation of sequential object orientation, we do not usually distinguish between inheritance and subtyping, but it is very helpful to emphasize the distinction between them in the issues of concurrent object orientation, for example, in the comprehension of *inheritance anomaly*[12].

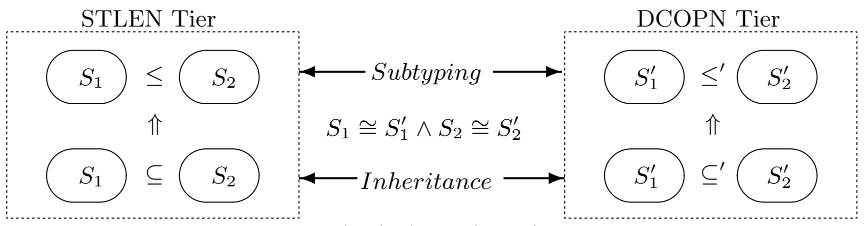
In the net-based multi-tier behavior inheritance modelling method of this paper, the subtyping relations are supposed to be preserved under the behavior preserving relations between two consecutive tier net models. We illustrate this in fig 4.1 by the two tier situation from the STLEN tier to the DCOPN tier, in which the relation $\leq$ can be preserved to the relation $\leq'$ under the restricted bisimulation relation $\cong$, where $\leq$ is the subtyping relation between STLCN nets, and $\leq'$ is the one between DCOPN nets. Each S_i represents a STLEN net, and each S_i' a DCOPN net. The relation $\preceq$ represents the least subtyping relation between two DCOPN Nets, which makes the interface of a subtype net usable in any context in which the interface of one of its supertype nets can be used.



$$S_2 \leq S_1 \wedge S_1 \cong S_1' \wedge S_2 \cong S_2' \wedge S_2 \preceq S_2' \Rightarrow S_2' \leq' S_1'$$

Fig. 4.1. Subtyping relation preserving

To satisfy $\preceq$ is a prerequisite in the definition of $\leq'$, which is the requirement to conform to the Principle of Substitutability [13]:*An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*



IF (1) $S_1' \subseteq' S_2' \Rightarrow S_1' \preceq S_2'$ THEN
(2) $S_1' \subseteq' S_2' \wedge S_1 \cong S_1' \wedge S_2 \cong S_2' \Rightarrow S_1 \subseteq S_2$

$$(S_1 \subseteq S_2 \rightarrow S_1 \leq S_2) \wedge S_1 \cong S_1' \wedge S_2 \cong S_2' \Rightarrow (S_1' \subseteq' S_2' \rightarrow S_1' \leq' S_2')$$

Fig. 4.2. Incremental inheritance preserving

One of the choices for the definition of $\leq$ is the one in [9,10], which is a bisimulation based on the grouping of S-elements and in terms of blocking or encapsulating actions, i.e., the observable actions special to the subtype objects are to be inhibited in the context of the supertype objects. As an example, an instance of Σ_2 will behave like an instance of Σ_1, if actions *lock* and *unlock* are blocked. The bisimulation relation may be $R = \{\langle\{empty\}, \{empty\}\rangle, \langle\{partial\}, \{partial\}\rangle, \langle\{full\}, \{full\}\rangle\}$, which is a relation from $\wp(\{empty,\ partial,\ full\})$, the powerset of all the S-elements in the S-elements group *state* in Σ_1, to $\wp(\{empty,$ $partial,\ full\})$, the powerset of all the S-elements in the S-elements group *state* in Σ_2. So we have $\Sigma_2 \leq \Sigma_1$. Usually a bisimulation relation in establishing a relation $\leq$ between STLEN systems is an union of such relations like R.

The definition of $\leq'$ in [10] is based on a bisimulation relation between the sets of *attribute predicates* of two DCOPN nets, also in terms of blocking actions. For example, a bisimulation relation from Σ_1' to Σ_2' may be $R' = \{\langle empty, empty\rangle, \langle partial, partial\rangle, \langle full, full\rangle\}$. Besides, to satisfy $\preceq$ is a prerequisite condition as stated above. In this example, the relation $\Sigma_2' \preceq \Sigma_1'$

simply demands that the *covariant* and *contravariant* rules as in [15] are hold for the methods *put* and *get*. These are evidently true. So we have $\Sigma_2' \leq' \Sigma_1'$.

For the definitions of $\leq$, $\leq'$, $\cong$ and $\preceq$ in [10], we can verify the proposition showed in fig 4.1. This proposition is required to be valid for every two successive specification tiers, which should be guaranteed when the multi-tier method is built.

4.3 Incremental Inheritance Preserving

Subtyping is a behavior preserving relation. Instead, inheritance is used for the code/specification reuse. Practically, to implement the behavior preserving while the code/specification is also easily reused, some incremental inheritance relation paradigms, capable of implementing the anticipant subtyping relations, need to be developed [8], the more the better. In our multi-tier inheritance modelling method, incremental inheritance relations are required to be preserved from the super-tier to the sub-tier, i.e., the THEN part in Figure 2 holds. One of the possibilities to obtain this is to ensure the IF part in fig 4.2 to be satisfied.

5 Modelling Steps

The following is a guide for the behavior inheritance modelling method in this paper:

(1) With the help of any OOA/OOD methodology, develop the behavior model of objects/classes using the net language STLEN and its available tools (to be developed). Label both the states (places) and actions (transitions), and the same time divide the states into groups according to the attributes.

(2) Develop DCOPN nets from STLEN ones by adding details, such as data types, constants, and attribute predicates. Build a map between the STLEN tier and the DCOPN tier in the same time when a DCOPN net is developed from its corresponding STLEN one. The map will be used in the verification of the restricted bisimulation relation $\cong$.

(3) Complete the interface specification for each DCOPN net. Verify the relation $\cong$.

(4) For the behaviour inheritance (subtyping) modelling, just consider the derived net in the STLEN tier first. Then develop the DCOPN net from the derived STLEN one according to (2) and (3). Don't forget that the interface specifications for each super DCOPN class net and its derived DCOPN class net, developed from STLEN one, have to satisfy the least subtyping relation $\preceq$.

(5) Develop and use incremental inheritance paradigms as many as possible. This may substantively save the work, as illustrated in fig 4.2.

(6) Changes in the modelling process are allowable, which is simplified in our method since the direct corrections in the DCOPN tier may be avoided.

Many aspects for the multi-tier methodology still need to be explored. Researchers who are interested in it may find many new topics about it.

References

1. Charles Lakos. From Colored Petri Nets to Object Petri Nets. Proceedings of 16th Int. Conference on the Application and Theory of Petri Nets, LNCS 935, Turin, Italy, Springer-Verlag,1995.
2. E.Batiston, A,Chizzoni, Fiorella De Cindo. Inheritance and Concurrency in CLOWN. Proceedings of the "Application and Theory of Petri Net 1995" workshop on "object-oriented programs and models concurrency", Torino, Italy 1995.
3. C.Sibertin-Blanc. Cooperative Nets. Proceedings of 15th Int. Conference on the Application and Theory of Petri Nets. LNCS 815, Zaragoza, Spain, Springer-Verlag, 1994.
4. D.Buchs and N.Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. Proceedings of 12th Int. Conference on the Application and Theory of Petri Nets, LNCS 524, Gjern, Denmark, 1991.
5. R.Valk. Petri Nets as Token Objects—An Introduction to Elementary Object Nets. Proceedings of 19th Int. Conference on the Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, 1998.
6. U.Becker and D.Moldt. Object-oriented Concepts for Colored Petri Nets. Proceedings of IEEE Int. Conference on System, Man and Cybernetics, vol. 3, 1993, pp 279–286.
7. A.Newman, S.M.Shatz, and X.Xie. An Approach to Object System Modelling by State-Based Object Petri Nets. Journal of Circuits, Systems and Computers, Vol.8, No.1,1998, pp 1–20.
8. W.M.P. Vander Aalst and J.Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. 18th Int. Conference on the Application and Theory of Petri Nets, LNCS1248, Toulouse, France, Springer-Verlag, 1997.
9. S. Wang, J. Yu and C. Yuan. A Pragmatic Behavior Subtyping Relation Based on Both States and Actions. To appear in Journal of Computer Science and Technology, vol.16,No.5, 2001.
10. S. Wang. A Petri Net Modelling Method for Concurrent Object-Oriented Systems. Ph.D thesis, Peking University, June 2001.
11. S.Christensen, N.D.Hansen. Colored Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. Proceedings of 14th Int. Conference on the Application and Theory of Petri Nets, LNCS 691, Chicago, USA, Springer-Verlag, 1993.
12. Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In Research Directions in Concurrent Object-oriented Programming, edited by G.Agha, P.Wegner and A.Yonezawa, The MIT Press, 1993, pp 107–150.
13. P.Wegner,S.B.Zdonik. Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like. In ECOOP'88 Proceedings. Lecture Notes in Computer Science 322, pp 55–77, Springer-Verlag, 1988.
14. C. Yuan: Principles of Petri Nets, Electronic Industry Press, Beijing, China, 1998.
15. P.America. Designing an Object-oriented Programming Language with Behavioural Subtyping. In Proc. of REX School/Workshop on Foundations of Object-oriented Languages(REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990, LNCS 489, Springer-Verlag, 1991, pp 60–90.

Specification Based Testing: Towards Practice[⋆]

Alexander K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
petrenko@ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract. Specification based testing facilities are gradually becoming software production aids. The paper shortly considers the current state of the art, original ISPRAS/RedVerst experience, and outlines the ways for further research and testing tool development. Both conceptual and technical problems of novel specification based testing technologies introduction are considered.

1 Introduction

The specification based testing (SBT) progressively moves from academic research area into real-life practice. The process of learning to handle SBT techniques has to overcome a lot of problems related to both technical and human/management facets of software development. Below we focus on technical problems, namely, on issues of specification and test suite development or, more specifically, we try to answer the questions:

- **Why limited use** — why SBT has not been widely introduced in industry practice yet?
- **What is the best** specification approach[1]?
- **Which feature first** — which SBT features should be provided first?

The work is mainly grounded on the practical experience of RedVerst[2] group of ISPRAS [27]. The experience was gained from industrial projects under contracts with Nortel Networks (http://www.nortelnetworks.com), Advanced Technical Services APS and research projects under grants of RFBR (http://www.rfbr.ru) and Microsoft Research (http://www.research.microsoft.com).

⋆ The work was partially supported by RFBR grant No. 99-01-00207 and Microsoft Research grant No. 2000-35.

1 We mainly consider the common Application Program Interface (API) testing and basically we do not focus on specific specification and testing methods intended for a specific kind of software like telecommunication, compilers, databases, and so on.

2 RedVerst stands for Research and development for Verification, specification and testing.

2 State of the SBT Practice — Why Limited Use?

State of the SBT art is very dynamic. During the last 5-6 years, a lot of sound results have been produced in research and industry spheres. The attention of academic researchers in formal specification is being shifted from analytical verification to problems of test generation from formal specification. The considerable number of testing tool producers have announced features related to SBT. The most progress has been achieved in specific areas like telecommunication protocol testing. There are successful attempts to deploy formal specification and SBT features for verification and validation of wide spectrum of software including API testing. But most commercial tools/technologies provide features for only partial specification (like assertions that describe only a part of functionality). The technologies do not provide instructive methodologies for specification and test design. Therefore, the deployment of the technologies faces troubles in scalability of the approach in real-life projects. Other important problems are the integration of SBT tools and techniques with widely used Software Development Environments (SDE) and the introduction of the new activities in the conventional SoftWare Development Processes (SWDP). No one SBT tool does provide a complete set of features that meet common requirements of specification and test designers. Instead, these tools try to suggest a "best and unique solution" . ADL/ADL2 story is a sad example of such approach. The ADL/ADL2 family of specification notations provided quite powerful and flexible features for specification of C, C++, and Java interfaces functionality. Nevertheless, test design problems (both methodological and tool support ones), especially in context of OO testing, were neglected. It seems that this reason has caused the refusal of ADL use in industrial practice. As promised, we restrict our consideration to only technical issues. However, the exhaustive answer to the above heading question has to involve, in addition, human and management facets (see more detailed consideration in [15,19]).

3 Specification Approaches — What Is the Best?

There are a few kinds of classification of specification approaches like model-oriented vs. property-oriented and state-based vs. action-based. To shortly review advantages and drawbacks of the specification approaches we will hold to following classification: *executable, algebraic* (usually, co-algebraic), *use cases or scenarios, and constraint specifications* (some specification kinds like temporal, reactive, etc. are outside of our consideration because their specifics).

Executable specifications, executable models. This approach implies developing a prototype to demonstrate feasibility and functionality of further implementation. The examples of the approach are SDL [20], VDM [19,23,26], explicit function definitions in RAISE [21]. Finite State Machines (FSM) and Petri nets could be considered as (more abstract) executable specifications too.

Algebraic specification provides a description of properties of some operations' compositions (serial, parallel, random, etc.). Usually this approach is tightly related to axiomatic approach [1,9]. SDL follows this approach to specify data types [1,20]; RAISE [21] provides quite powerful facilities for axiomatic specification.

Use case/Scenario based specification approach suggests considering the scenarios of use instead of properties of the implementation. The approach is developed and propagated by OMG/UML community [28]. SDL community uses MSC [14] notation for scenario description. The informative review of the scenario-based testing techniques is presented in [19].

Constraint specification implies a description of data type invariants and pre- and post-conditions for each operation (function, procedure). There are specific techniques for OO classes and objects specification. The constraint specification approach is followed by VDM [3,19], Design-by-contract in Eiffel [24], implicit function definition style in RAISE, iContract [10], ADL [22].

From the testing perspective, the advantages and the drawbacks of the specification approaches could be evaluated by simplicity of the specification development and simplicity of the test derivation from the specification. For example, algebraic and FSM-like specifications are very suitable for the test sequence generation including the case of concurrent and distributed software. However, the approach provides very restricted opportunities in test oracle[3] specify their software using only algebraic or FSM approach. Besides, the algebraic specification is a non-scalable approach. Such specifications for a toy example are very short and attractive, however, as the size increases, the understandability of the algebraic specification drastically drops (however, there exists a society of experts in algebraic specification who do not share this opinion).

So, in short, the heading question answer is "there is no the best specification approach". Specification and test designers need good composition of specification techniques. One example of such composition is a combination of constraint and FSM specification used in [7,12]. Another example is SDL/MSC/TTCN composition that is widely used in SDL users' society.

4 RedVerst Experience and Lessons Learned

Before answering third above declared question "Which feature first?" let's dwell on the lessons learned from the RedVerst experience. ISPRAS organized the RedVerst group accepting the challenge of Nortel Networks in 1994. The original goal of the group was developing a methodology applicable to conformance testing of API of Nortel Networks proprietary real-time operating system. By the end of 1996, RedVerst has developed the KVEST methodology [7,8,16,25], the specifications and the tools for test generation and test execution. The RAISE Specification Language (RSL) [21] was used for specification. The KVEST included techniques for automatic and semi-automatic test generation, automatic test execution and test result analysis and reporting. The techniques were oriented onto use in real-life processes, so, some practical requirements must be met like: fault tolerant testing, fully automatic re-generation, re-run of the tests

[3] *Test oracle* is a decision procedure that automatically compares actual behavior of a target program (outcome of a target operation) against its specification [17].

and test result analysis. The total size of KVEST formal specifications now is over 200 Kline. Six patent applications have been filed based on the KVEST research and development experience, a few patents have been taken out.

The most valuable KVEST solutions were as follows:

– A few kinds of test scenario schemes. The simplest scheme was intended for separate testing pure functions (without side effect) and allowed fully automatic test generation. The most complex schemes allow testing parallel execution of software like resource managers and messaging systems.
– Enhanced test generation technique that allows excluding from consideration the inaccessible and redundant test situations.
– Programming language independent technology scheme for test generation.
– Automatic integration of generated and manually developed components of test suites for semi-automatic test generation. The technique allows avoiding any manual customization during repeated re-generations of test suites. The feature is valuable for both test design and regression testing periods.

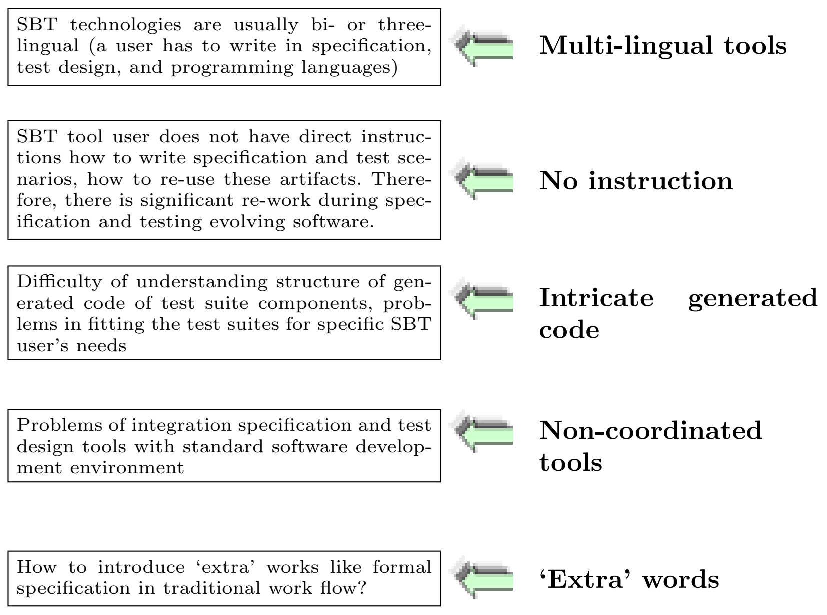
Up to now KVEST users have gained successful experience in verification of the following kinds of software.

– Operating system kernel and utilities
– Fast queuing systems for multiprocessor systems and for ATM framework
– Telecommunication protocols as a whole and some protocol implementation subsystems like protocol parsers.

Software verification processes. The KVEST has been applied in two kinds of software verification processes. First one is “Legacy reverse-engineering and improving process” and second one is “Regression testing process”.

In addition, the RedVerst has suggested a specific SWDP called *“co-verification”* process. The process unites the target software design with the formal specifications and the test suites development in a concurrent fashion. One of the valuable advantages of the process is the production of the test suites before the target implementation is completed. Another important benefit of the process is a clear scheme of cooperative work of architects, designers and test designers. The “co-verification” advantages provide good opportunities for early detection of design (the most costly) errors.

Lessons learned. On the one hand, the KVEST has demonstrated feasibility of SBT use in industrial applications. On the other hand, the KVEST has been successfully deployed as technology for only regression testing. The customer has not undertaken roles of specification and test designers yet. Usually the similar problems of technology deployment are explained by resistance of users and managers in accordance to formal techniques as a whole. It is true, but there exist important reasons of the “resistance” . The reasons are common for KVEST and for many other SBT technologies. The most significant reasons are as follows:



SBT technologies are usually bi- or three-lingual (a user has to write in specification, test design, and programming languages)

Multi-lingual tools

SBT tool user does not have direct instructions how to write specification and test scenarios, how to re-use these artifacts. Therefore, there is significant re-work during specification and testing evolving software.

No instruction

Difficulty of understanding structure of generated code of test suite components, problems in fitting the test suites for specific SBT user's needs

Intricate generated code

Problems of integration specification and test design tools with standard software development environment

Non-coordinated tools

How to introduce 'extra' works like formal specification in traditional work flow?

'Extra' words

Fig. 1. SBT problems

5 Next Step — Which Feature First?

To overcome these five problems, the five following solutions are suggested.

"Multi-lingual tools" problem. It is the first well-recognized problem in SBT: what specification language (notation) should be chosen in a practical project. There are two main alternatives in the notation choice: the formal specification languages (like classical ones) and **extensions of usual programming languages**. Both alternatives have advantages and drawbacks. The KVEST technology followed the first way and has demonstrated feasibility of the approach. However, now it's becoming evident that the second way promises more advantages in the real-life industry context. The main argument for programming language extension vs. using formal specification language is the evident advantage of a monolingual system against a bi- or multi-lingual system. In addition, there are problems of involving software engineers into study and training in formal specification languages.

The idea of programming language extension for specification purpose is not a novel one. The similar suggestions were discussed by B.Liskov, D.Parnas, and others in early 1970s. In the mid of 1990s C, C++, and Java were extended by joint X/Open Company Ltd. and the Sun Microsystems, MITI's Information-technology Promotion Agency group [22], Java by R.Krammer [10]. Eiffel [24], VDM-SL and VDM++ had originally facilities both for programming (prototyping) and for constraint specification.

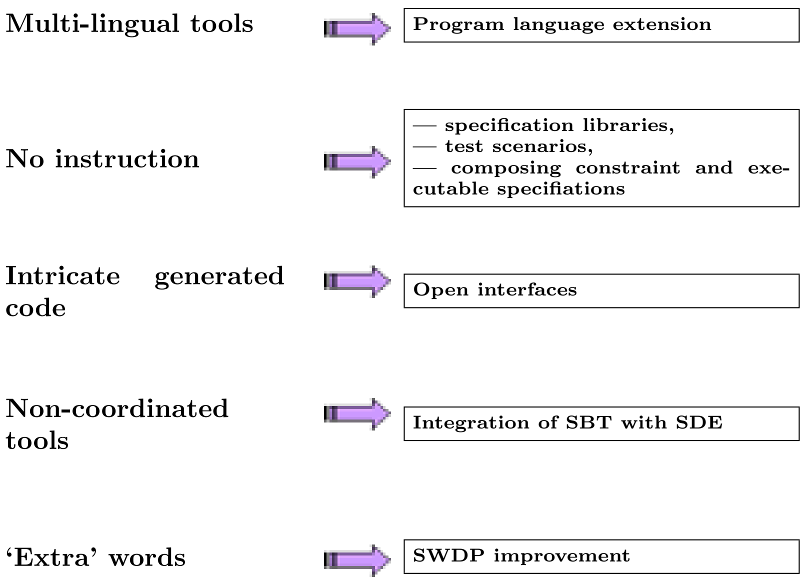


Fig. 2. UniTesK solutions

Success of these extensions is quite restricted. Some of the tools/technologies are used only as in-house tools, others are mainly used in academic area. The reason of the obstacle is weakness and incompleteness of features provided to a practical specification and test designer. As an example, in more details the drawbacks of ADL and iContract are described in another paper presented in the proceedings and on the RedVers web-site [4,27]. RedVerst has designed UniTesK concept of SBT using programming language extension. The concept is presented in [15].

"No instruction" problem. The tutorials, monographs, and manuals are necessary but not sufficient materials for SBT propagation. In addition to these materials, the software engineers need examples, prototypes, **libraries of specifications**. The OO approach opens new opportunity for architecture of these libraries [4,15]. Since specifications are usually more abstract than implementation, so re-use of the specifications could be simpler. This opportunity is noted by Eiffel society [24], but they use it only for rapid prototyping. An additional valuable advantage caused by introduction of formal specification in a software development process is the **test suite component re-use**, because these components are generated from **re-usable specifications**. RedVerst has developed the techniques for test suite re-use. The first one is intended for C-like software and uses template-based technology. The UniTesK approach expands the area of re-usable components and provides the OO techniques for representation of storing artifacts and integration of the handmade and the generated artifacts into the ready for use OO test suites.

As mentioned above, **executable specifications**, including FSM like specifications, are quite suitable for test sequence generation but have restricted possibilities for test oracle generation. There is an evident idea: to unite executable and constraint specifications to gain advantages of both these approaches. However, two obstacles prohibit from the union. First, it is doubling effort of specification design, and, second, there is a certain risk in developing the inconsistent parts of specifications. Some researchers try to derive executable specification from constraint specification automatically [12,13]. The idea seems to be quite attractive but cannot be applied to any kind of real-life software. RedVerst has developed the techniques for replenishment of constraint specification with implicit FSM specifications. This technique is briefly described below (see section 6) and in more details in [1,5,6]. New kinds of FSMs differ in degree of determinism and timing characteristics of reaction appearing. The variety of FSMs allows generating the test sequences for wide spectrum of software including distributed and real-time applications. FSM-based techniques allow generating an exhaustive test (in sense of the model). Sometimes (for example, for debugging) it is desirable to use non-exhaustive but some specific tests usually based on use cases or **test scenarios**. A union of scenario approach and FSM-based approach is used in UniTesK [4]. The technique allows describing the main idea (scenario) of a test case family and generating several test cases (using the implicit FSM technique) that belong to this family.

"Intricate generated code" problem. It is a common problem of the tools generating code from some sources; the intricate generated code is too complicated for understanding and debugging. A prospective solution is to design an **open OO test suite architecture**, where the generated and handmade artifacts are stored separately, in different classes, but are closely and naturally linked by means of usual relations used in OO design and programming. UniTesK presents an example of such OO test suite architecture [4,15].

"Non-coordinated tools" problem. The UniTesK dream is integration with arbitrary SDE based on some standard interfaces. UniTesK requirements in this case are as follows. SDE should provide facilities for:

– SBT tools invocation from the SDE
– synchronization of the windows related to SBT input/output
– key words/syntax setting
– diagnostic messaging.

"'Extra' works" problem. Introduction of SBT implies appearance of new activities, roles, and artifacts in SWDP. The specifications could be presented as informal (for example, draft functional requirement specification), semi-formal (like UML use cases), and formal specifications. The new artifacts raise the necessity in new personnel, techniques, and tools - negative consequences. They allow well-organized and computer-aided requirements tracking, rapid prototyping, and test and documentation generation - positive consequences. Therefore,

to take an advantage of SBT, an organization should invest some additional effort to compensate possible negative consequences of this prospective technology (it is true for any novel technology).

6 Go into UniTesK

UniTesK is intended for SBT. Therefore, on the one hand, it is implied that a specification or a model exists, but on the other hand, testing must check the implementation not only on the robustness, but also on the conformance with the specification [4]. A specification could be written, in principle, in any suitable specification (modeling) language. UniTesK proposes to use for this purpose not conventional specification languages, but extensions of usual programming languages. It should facilitate introduction of UniTesK technologies in the industry practice.

Let's consider the basic components of the source for test generation and the structure of test suites and test harness [5] used in UniTesK. We begin with relatively simple kind of software - procedures (subroutines, functions, methods, etc.) that allow separated testing. I.e. no action of the system under test (SUT) is needed to set the SUT in the initial state, to form input test parameters, call the procedure under test, and evaluate its result. Most of the mathematical and string manipulation functions are examples of such procedures.

What tasks should be solved by a test system in separated testing? They are as follows:

- conduct partition analysis of the input space to be able to define the equivalence classes of the input test data;
- form input test parameters (or generate a program that will form the test parameters in dynamics);
- if the number of generated test cases is too large, test system should provide filters for test case selection (based on the above-mentioned equivalence class definitions);
- invoke (probably, repeatedly, in a loop) the procedure under test with corresponding parameters;
- check correctness of the procedure result;
- collect trace to get information on the detected errors and the achieved test coverage.

[4] Notice that the two tasks are simultaneously an advantage and a drawback. SBT allows to bridge functional requirements and implementation behavior, it is a positive facet. At the same time, the necessity of specification development extends time length and raises the cost of the software development. In addition, the specification needs personnel with specific skills.

[5] Test harness in UniTesK consists of a test suite (specification and implementation dependent part of test program) and an invariant part of the test program (run-time support or operation environment of the test suite), that could be used for testing wide spectrum software on the platform.

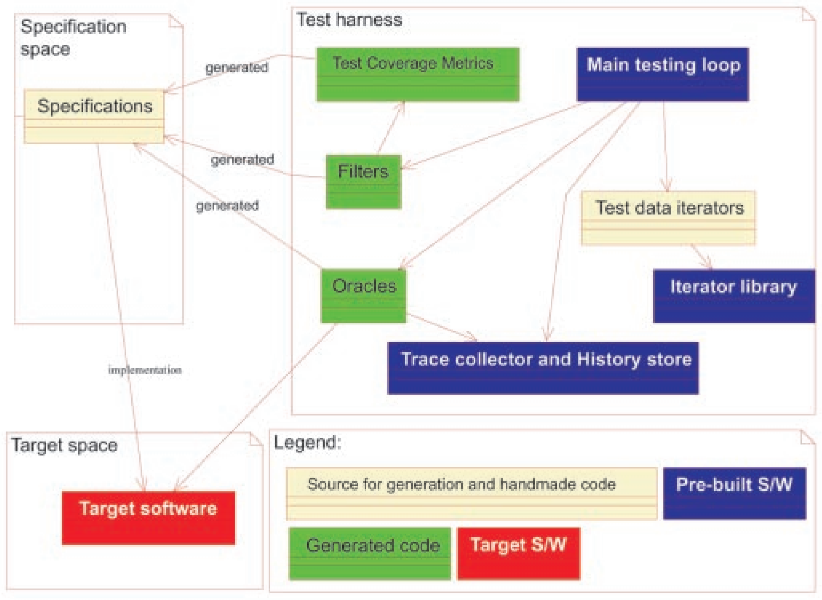


Fig. 3. UniTesK test program architecture in the case of separated testing

Is it possible to generate fully automatically a test suite that performs all the tasks? Sometimes it is possible! Thus, about 40% of procedures of the kernel interface layer of Nortel Networks switch OS allowed separated testing [7]. For about half of them the test suites were generated fully automatically. For the rest of the procedures, data iterators were written "by hands". The handmade work appeared to be so simple, that the benefits of automatic generation are not too valuable. So, both "automatic" and "handmade" ways are possible, but in the second case we should be able to integrate the generated and the handmade components. A reader should keep in mind that "test generation", in fact, is a phase of the industrial process that implies repeatable re-generation, re-integration, re-execution, and so on. So, the problem of integration of handmade and generated components should be solved very carefully and efficiently. The integration requires elaborating the structure and the relations between all components of a test suite.

Therefore, although sometimes a test suite can be generated fully automatically, in general case we have to generate a framework that includes common or pre-built parts, generated parts, and slots for handmade components. The test suite is integrated into an operating environment. The operating environment is the invariant part of test programs. Together a test suite and the invariant part are named the "test harness ". Our experience showed that this way is quite acceptable in industrial practice, because it provides flexibility and decreases test designer's effort.

Testing of procedure groups, where the procedures share common data and/or testing OO classes/objects, raises new problems. So, the architecture and the functions of test system is becoming more complicated. What are these new problems and tasks? Nominally, there is only one task: to define an order on invocations of the target functions.

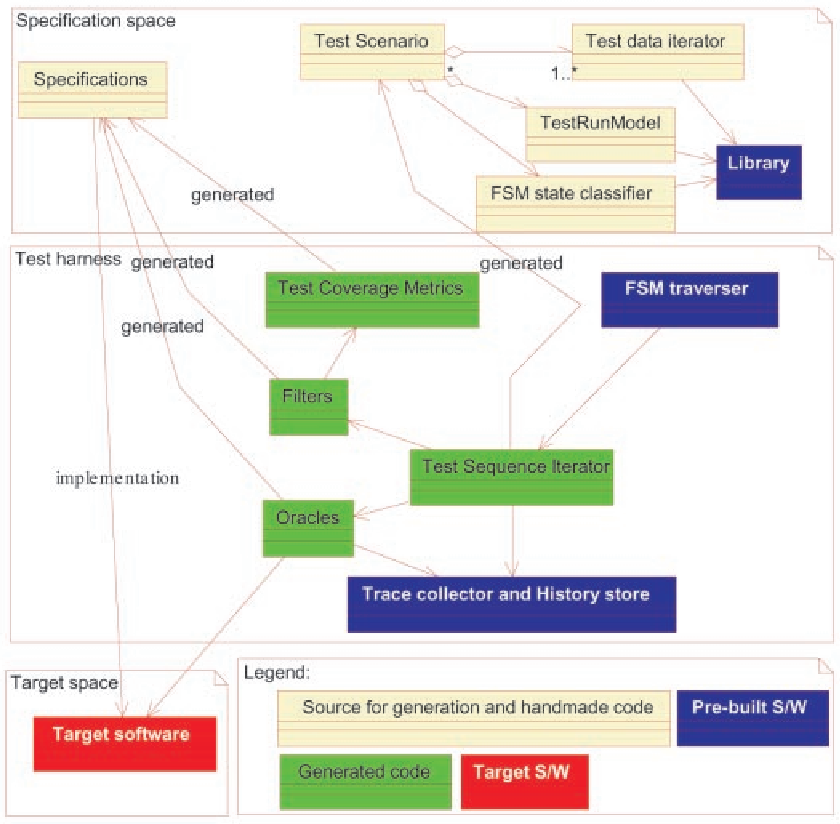


Fig. 4. UniTesK test program architecture. General case

An idea of FSM traversing underlies the UniTesK test sequence generation scheme. In principle, if we digress from the SBT context, then the implementation itself could play the role of the FSM. In this case, a state of global variables corresponds to the FSM state, and invocations of the target functions correspond to transitions between the states. However, firstly, FSM of a real-life software would be too big and complicated, and, secondly, we are interested in extraction of FSM from specification or providing a relation between the FSM and the specification. So, we have to introduce a more "simple" FSM, which is a factorization (reduction) of the FSM given by the specification. As a result of factorization, the states of the reductive FSM correspond to classes of the source specification states and the transitions of the reductive FSM correspond to classes of transitions in the source specification FSM (function invocations) or to a chain of transitions (sequence of invocations). UniTesK allows implicit description of FSM. Such description does not directly enumerate all states and all transitions. It turned out that for states we can define only a Boolean function that compares two states; for transitions user should define a function-iterator which builds the next invocation based on the history stored in the current node of the FSM. Our experience shows that such description is more simple (shorter and needs less effort) than an explicit description used, for example, in [12]. All changes in the test suite architecture can be seen in Figure 4. Thus, FSM traverser occupies the place of the main testing loop in Figure 3. The traverser is a unified program that can traverse (make transitions step-by-step to gradually cover all states and all transitions in the FSM) any FSM given by a test scenario. The particularities of the FSM, its relation to the target system (its mapping to the target system interface) are presented in Test Scenario. The Test Scenario is a base for Test Sequence Iterator. A Test Scenario, on the one hand, is bound to the target interface specification (it takes into account the target function/methods names, number and types of parameters and so on). On the other hand, the logic of the Test Scenario could be independent of the logic and the purpose of the target system. In the later case, test designer can use the library TestRunModel, as well as the library iterators and classifiers.

FSM kinds. The proposed architecture of test suite is quite general. In particularly, it allows different kinds of FSM: deterministic, non-deterministic, partial cases of non-deterministic FSM. This opportunity has been used. So, a kind of non-deterministic FSM was used for testing storage allocation system. In research project under grant of Microsoft Research [29] on testing of IPv6 protocol implementation, we used so-called "FSM with delayed reactions ". This kind of FSM meets requirements of a wide range of distributed and telecommunication systems. As distinct from usual FSM, this kind of software has a non-usual particular feature - for most states, a list of permissible reactions on the stimuli is known, but we do not know the time of the reaction and the order of expected reactions. So, stimuli and reactions form some intricate combinations and only a partial ordering is defined.

The project has shown the flexibility of the UniTesK test suite architecture. To introduce such kind of FSM, we changed nothing except for the FSM traverser. The delayed reactions were mapped into oracles. These oracles operate with so-called hidden stimuli. The purpose of these oracles is to check whether a corresponding reaction has arrived yet and, if so, whether the order of the reaction is correct or not.

7 UniTesK Foundation and UniTesK Program

The fundamental distinction of UniTesK in formal method applications is the rejection of classical specification languages though they provide abstraction features and strong semantics. Are UniTesK specifications and tests built on sand or not?

Traditionally, specification and modeling pay very serious attention to strong semantics definition on the notation used. Otherwise, as researchers say, it is impossible to treat and interpret the specifications (see, for example [30]). Notice, the above terms “treat” and “interpret” imply here an automatic analysis, reasoning, analysis of conformance between specification and implementation. In this context, they are right. But Yu.Gurevich [31] notices that requirements of modeling and reasoning are very different and have different effects on specification languages. Furthermore, we should distinguish between requirements caused by specification/description/modeling and requirements caused by a certain kind of reasoning.

The analytical verification makes very strong demand to semantics definition of a specification language. Should SBT share this demand? It seems, no. Testing practice, as well as programming as a whole, does not meet any fundamental trouble caused by a vagueness of some programming language constructs and or test design languages. Why?

The answer is simple: programmers avoid “dangerous” constructs and obscure operating situations. Following this principle, we can build specification extensions of programming languages and use these extensions. In other words, we do not require a specification extension to be more rigorous then a basic programming language.

UniTesK represents a program consisting of a family of industrial and research activities. The common goal of this program is to develop and deploy in the real-life practice the techniques and tools supporting the SBT. As mentioned above, the first step on this way is development and introduction of specification extensions of programming languages together with tools for test generation, integration of the tools with the usual SDEs. Now we have developed extensions and tools for Java and C. VDM++TesK add-on tool has been developed for VDMTool box (here we consider VDM++ as a programming language for rapid prototyping). In the nearest future, UniTesK will be implemented for C++, C#, and VDM-SL platforms.

8 Conclusion

The paper presents an outline of current state of the art and prospective solutions of SBT problems. We focus on API specification testing because it is the base and most uniform level of software interfaces. This short review of SBT techniques did not consider whole variety of techniques known academic area. Our attention was only paid to the approaches have been used in real-life SWDP.

There is no any unified approach for specification and SBT and inside each of these approaches there is no any unique tool that performs all necessary actions. It is rather well. However, the known research results and commercial tools have shown feasibility of SBT approach in real-life application of arbitrary complexity. To be introduced in practice any technology must provide at least minimal set of features that meet the most needs, it is so-called critical mass. Above "Next step" solutions outline the critical mass. The era of toy examples and pioneer SBT projects is finishing.

References

1. Abstract Syntax Notation One (ASN.1) standard. Specification of Basic Notation. ITU-T Rec. X.680 (1997) — ISO/IEC 8824-1:1998.
2. S. Antoy, D. Hamlet. Automatically Checking an Implementation against Its Formal Specification. IEEE trans. On Soft.Eng, No. 1, Vol. 26, Jan. 2000, pp. 55–69.
3. D. Bjorner, C. B. Jones (editors). Formal Specification and Software Development. — Prentice-Hall Int., 1982.
4. I. B. Burdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. J@va : extension of Java for real-life specification and testing. — In this volume. — P.303–309.
5. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin, A. V. Maximov. Testing Programs Modeled by Nondeterministic Finite State Machine. — (see [27] white papers).
6. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Using Finite State Machines in Program Testing. "Programmirovanije", 2000, No. 2 (in Russian). Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 61–73 (English version).
7. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. Proceedings of World Congress on Formal Methods, Toulouse, France, LNCS, No. 1708, 1999, pp. 608–621.
8. I. Burdonov, A. Kossatchev, A. Petrenko, S. Cheng, H. Wong. Formal Specification and Verification of SOS Kernel. BNR/NORTEL Design Forum, June 1996.
9. R.-K. Doong, P. Frankl. Case Studies on Testing Object-Oriented Programs, Proc. Symp. Testing, Analysis, and Verification (TAV4), 1991, pp. 165–177.
10. R. Kramer. iContract — The Java Design by Contract Tool. Fourth Conference on OO technology and systems (COOTS), 1998.
11. S.-K. Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications (http://svrc.it.uq.edu.au/Bibliography/bib-entry.html?index=851).
12. L. Murray, D. Carrington, I. MacColl, J. McDonald, P. Strooper. Formal Derivation of Finite State Machines for Class Testing. In: Lecture Notes in Computer Science, 1493, pp. 42–59
13. K. Lerner, P. Strooper. Refinement and State Machine Abstraction. — SVRC, School of IT, The University of Queensland, Technical report No. 00-01. Feb. 2000 (http://svrc.it.uq.edu.au/).
14. Message Sequence Charts. ITU recommendation Z.120.
15. A. K. Petrenko, I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Experiences in using testing tools and technology in real-life applications. — Proceedings of SETT'01, India, Pune, 2001.
16. A. Petrenko, A. Vorobiev. Industrial Experience in Using Formal Methods for Software Development in Nortel Networks. — Proc. Of the TCS2000 Conf., Washington., DC, June, 2000.
17. D. K. Peters, D. L. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Trans. on Software Engineering, Vol. 24, No. 3, March 1998, pp. 161-173.

18. N. Plat, P. G. Larsen. An Overview of the ISO/VDM-SL Standard. SIGPLAN Notices, Vol. 27, No. 8, August 1992.
19. J. Ryser, M. Glinz. SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. — ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.03.pdf
20. Specification and Design Language. ITU recommendation Z.100.
21. The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
22. http://adl.opengroup.org/
23. http://www.csr.ncl.ac.uk/vdm/
24. http://www.eiffel.com
25. http://www.fmeurope.org/databases/fmadb088.html
26. http://www.ifad.dk
27. http://www.ispras.ru/~RedVerst/
28. http://www.omg.org/uml/
29. http://research.microsoft.com/msripv6/
30. Michael Barnett and Wolfram Schulte. The ABCs of Specification: AsmL, Behavior, and Components (Draft of paper submitted to Informatica.) http://research.microsoft.com/foundations/comps.ps
31. Yuri Gurevich. Private communication.

Java Specification Extension for Automated Test Development

Igor B. Bourdonov, Alexey V. Demakov, Andrew A. Jarov,
Alexander S. Kossatchev, Victor V. Kuliamin, Alexander K. Petrenko, and
Sergey V. Zelenov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
`{igor,demakov,jandrew,kos,kuliamin,petrenko,zelenov}@ispras.ru`
`http://www.ispras.ru/~RedVerst/`

Abstract. The article presents the advantages of J@va, a specification extension of the Java language, intended for use in automated test development. The approach presented includes constraints specification, automatic oracle generation, usage of FSM (Finite State Machine) model and algebraic specifications for test sequence generation, and specification abstraction management. This work stems from the ISPRAS results of academic research and industrial application of formal techniques [1].

1 Introduction

The last decade has shown that the industrial use of formal methods became an important new trend in software development. Testing techniques based on formal specifications occupy a significant position among the most useful applications of formal methods. However, several projects carried out by the RedVerst group [3,12] on the base of the RAISE Specification Language (RSL) [6] showed that the use of specification languages like RSL, VDM or Z, which are unusual for common software engineer, is a serious obstacle for wide application of such techniques in industrial software production. First, the specification language and the programming language of the target system often has different semantics and may even use different paradigms, e.g., one can be functional and the other can be object-oriented, so a special mapping technique must be used for each pair of specification and target language. Second, only developers having special skills and education can efficiently use a specification language. The possible solution of this problem is the use of specification extensions of widely used programming languages.

This article presents J@va – a new specification extension of Java language. Several specification extensions of programming languages and Java in particular already exist. ADL [5,7] and iContract [8,9] are the most known of them. A few extensions have been used in industrial projects. Why do we invent a new one?

Our experience obtained in several telecommunication software verification projects shows that the formal testing method used in industry should not only allow automated test generation but also possess features such as clear modularization, suitable abstraction level management, separate specification and test design, and the support of test coverage estimation based on several criteria [14]. These subjects did not receive sufficient attention in ADL and iContract languages and test development technologies based on them. The absence of integrated solution explains the limited use of specification based methods in general and these languages and related tools particularly.

Every industrial test development technology should provide the answers on the following three questions.

– How to determine whether the component of the target system behaves correctly or not?
– How to determine the set of test cases, which makes the test complete in the sense that any additional test case can not add any important information? And how to estimate the results of the test, which does not contain all of these test cases for some reasons?
– How to organize the sequence of target operations calls during the test, *the test sequence*, to obtain the necessary test cases in the most effective way?

The first problem is usually solved with the help of the component's *oracle,* which can be generated from the specification of the component. Although oracle generation techniques are known (see, for example, [3,4,5]), they are not widely used in industrial projects. KVEST project [3] performed by our group is the largest example of such a project in the European formal methods database [2]. We do not consider the issues of automated oracle generation in this paper to comply with size restrictions. We refer the interested reader to the previously mentioned works [3,4,5].

The solution of the second problem is very important for the industry. Usually the solution is based on *test coverage criterion,* which gives the numerical measure of the test effectiveness. Coverage criteria based on the structure of the target code are widely known and used in the industry, but they are not sufficient. Such criteria show what part of target code is touched by the test. The important issue is also what part of functionality is touched by it. This problem can be solved with the help of *specification based coverage criteria.*

The third problem mentioned above is addressed by FSM based testing technique used in our approach called UniTesK technology (see [11] for details). The approach uses FSM model of the unit under test. Such a model is developed on the base of the coverage criterion chosen to obtain. Then, the FSM developed is traversed, and, so, the target coverage is achieved. We call the description of this model *the test scenario.* Test scenarios directly deal with test design while specifications describe the abstract functionality of target system.

2 Key Features of J@va Approach

In this section we present the goals of J@va design and J@va key features, explain their advantages, and compare them with ADL and iContract.

During the design of J@va language we tried to preserve the main features of our test development method, which is based on several previous successful projects (see [3]). These features are as follows.

– Automatic generation of test oracles on the base of specifications presented as pre- and postconditions of target operations.
– The possibility to define test coverage metrics and automatic tracking of coverage obtained during the test.
– Flexible FSM based testing mechanism.
– Dynamic test optimization for the target coverage criterion.

Along with that we must simplify and clarify the method to make it applicable in the software industry, not only in the research community.

Below we pay more attention to features not supported or supported insufficiently by J@va contenders. In particular, we do not consider the general software contract specification approach used in all mentioned languages [8,13] and methods to specify exceptional behavior. Parallel processing specification and testing methods are also out of the scope of this article.

Specification of object state. J@va specifications are structured similar to Java code. The specification of the behavior of some component is presented as a specification class, which can have attributes that determine the model state, *invariants* representing the consistency constraints on the state of an object of this class, and specification methods representing the specifications of operations of the target component. Specification method can have pre- and postcondition. In ADL and iContract specifications are also presented as pre- and postconditions, but they have no special constructs for state consistency constraints. The same effect can be obtained only by including such constraints into pre- and postconditions of all class methods.

Axioms and algebraic specifications. J@va provides constructs to express arbitrary properties of the combined behavior of target methods in an algebraic specification fashion. The semantics of J@va axioms and algebraic specifications is an adaptation of the semantics of RSL ones [6]. Axioms and algebraic specifications serve as a source for test scenarios development – they are viewed as additional transitions in the FSM testing model. During testing we call the corresponding oracle for each method call in an axiom and then check the global axiom constraint. Similar constructs can be found only in specification languages and are absent in specification extensions as ADL and iContract.

Test coverage description. This is an essential feature for testing and software quality evaluation. Test coverage analysis also helps to optimize the test sequence dynamically by filtering the generated test cases, because usually there is no need to perform a test case that does not add anything to the coverage already obtained. Each coverage element can have only one corresponding test case. The coverage consisting of domains of different behavior, called the *specification branch coverage,* can be derived automatically from the J@va postcondition structure. J@va also has several special constructs for explicit test coverage description. The explicit coverage description and functionality coverage derivation allow providing fully automatic test coverage metrics construction and test coverage analysis. Neither ADL nor iContract has facilities for test coverage description and analysis.

Abstraction level management. The ability to describe system on different abstraction levels is very important both in forward and reverse engineering of complex systems. The support of abstraction level changing allows developing really implementation-independent specifications, whether we follow top-down design or bottom-up reverse engineering strategy. In J@va, specifications and source code are fully separated. Their interaction is provided by a special *binding code.* This code performs synchronization of the model object state with the implementation object state and translates a call of model method into a sequence of implementation methods invocations. It is necessary, because test sequence is defined on the model level in our method. This approach allows using one specification with several source code components and vice versa, it also ensures the modularity of specifications and makes possible their reuse. No other of known Java specification extensions provides such a feature. Larch [10] provides the infrastructure the most similar to the J@va one but supports only two-level hierarchy.

Test oracle generation. This is a standard feature of specification extensions intended to be used for test development. J@va, as ADL and iContract, supports automatic generation of test oracles from the specifications.

Test scenarios. Test scenarios provide the test designer with a powerful tool for test development. The scenarios can be either completely user-written or generated on the base of once written templates and some parameters specified by test designer. In general, a J@va scenario defines its own FSM model of the target system, called *the testing model.* A scenario defines the state class for this model and the transitions, which must be described in terms of sequences of target method calls. The testing model should represent a FSM, which can be obtained from the FSM representing the target system by removing some states and transitions, combining a sequence of transitions into one transition and subsequent factorization. One can find details of this approach, some methods and algorithms of testing model construction in [11], where they are formulated in terms of FSM state graph properties.

In a more simple case, test scenario represents the sequence of tested operation calls that can lead to some verdict on their combined work. The test constructed from such a scenario executes the stated sequence and assigns the verdict; it also checks the results of each operation with the help of the operation's oracle.

J@va allows the use in scenarios such constructions as iterations, nondeterministic choice and serialization. Iterations help to organize test case generation for one target operation. J@va test scenario is represented as a class with special methods – so-called scenario methods, which represent the transitions of the model.

Among existing Java extensions, only ADL provides some constructs for test case generation. However, complex tests, e.g. for a class as a whole, have to be written entirely in the target programming language. An essential shortcoming of this approach is the lack of state-oriented testing support that forces the test designer to spend considerable effort to ensure the necessary test coverage.

Open OO verification suite architecture. The verification suite consists of specifications, test scenarios, binding code, and Java classes generated from the specifications and the test scenarios. The set of classes and relations between these classes and between verification classes and target Java classes are well defined. The architecture is described in UML and is easy to understand by any software engineer having experience in design and development using Java language. The openness of the architecture does not mean necessity of the generated code customization for optimization or other purposes. There are other well-defined flexible facilities for fitting the verification suite. However the openness significantly facilitates the understanding and the use of the technology as a whole. ADL and iContract users could read (and reverse engineer) generated code, however the structure of generated test harness is considered a private issue of ADL/iContract translator and can be changed at any time.

Example of J@va specifications. Here we give an example of J@va specifications for bounded stack class with non-null elements. This example demonstrates some of the features itemized above.

```
specification package ru.ispras.redverst.se.java.examples.stack;
import java.util.Vector;

class StackSpecification {
  static public int MAX_SIZE = 2048;
  public Vector items = new Vector(MAX_SIZE); // model state

// object intergity constraint
  invariant I1()
  {
    return items.size() >= 0 && items.size() <= MAX_SIZE;
  }

// specification of pop() operation
  specification public synchronized Object pop()
    updates items.? // changes data available through items field
  {
    pre { return items.size() != 0; }
    post
    {
      branch "Single branch"; // defines single coverage element
      Vector old_items = items.clone();
    // method identifier refers to the result of operation
      old_items.addElement(pop);
    // @<expression> denotes the value of expression in pre-state
      return old_items.equals(@items.clone());
    }
  }
// specification of push() operation
  specification synchronized void push(Object obj)
    reads obj, updates items.?
  {
    pre { return obj != null && items.size() != MAX_SIZE; }
    post
    {
      branch "Sinlge branch";
      Vector new_items = @items.clone();
      new_items.addElement(obj);
      return items.equals(new_items);
    }
  }

// algebraic specifications
  equivalence synchronized Object push_pop(Object obj)
  {
    pre { return items.size() != MAX_SIZE; }
    alternative { push(obj); return pop(); }
    alternative { return obj; }
  }

  equivalence synchronized void pop_push() {
    pre { return items.size() != 0; }
    alternative { push(pop()); return; }
    alternative { return; }
  }
}
```

3 Conclusion

To become applicable in industrial software production, an automated test development technology must support a set of features that constitute something like a critical mass. The critical mass should be not too huge to be introduced in real-life software engineering and at the same time it should be sufficient for usual needs of software engineers. The J@va tries to achieve this goal. More detailed description of J@va and J@va based technology are presented on our web site [1].

References

1. http://www.ispras.ru/~RedVerst/
2. http://www.fmeurope.org/databases/fmadb088.html
3. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. *FM'99: Formal Methods. LNCS,* volume 1708, Springer-Verlag, 1999, pp. 608–621.

4. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering,* 24(3):161–173, 1998.
5. M. Obayashi, H. Kubota, S. P. McCarron, L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.xopen.org/exgr/icse/icse98.htm
6. The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
7. http://adl.xopen.org
8. R. Kramer. iContract – The Java Design by Contract Tool. // 4-th conference on OO technology and systems (COOTS), 1998.
9. http://www.reliable-systems.com/tools/iContract/iContract.htm
10. J. Guttag et al. The Larch Family of Specification Languages. // IEEE Software, Vol. 2, No. 5 (September 1985), pp. 24–36.
11. I. Bourdonov, A. Kossatchev, V. Kuliamin. Using FSM for Program Testing. *Programming and Computer Software,* Official English Translation of *Programmirovanie,* No. 2, 2000.
12. A. Petrenko, I. Bourdonov, A. Kossatchev, and V. Kuliamin. Experiences in using testing tools and technology in real-life applications. Proceedings of SETT'01, India, Pune, 2001.
13. B. Meyer. Object-Oriented Software Construction. Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1997.
14. A. K. Petrenko. Specification Based Testing: Towards Practice. In this volume. P.289–302.

Specification-Based Testing of Firewalls*

Jan Jürjens[1] and Guido Wimmel[2]

[1] Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, Great Britain
tel. +44 1865 284104, fax +44 1865 273839, `jan@comlab.ox.ac.uk`

[2] Department of Computer Science, Munich University of Technology
TU München, 80290 München, Germany
tel. +49 89 289 28362, fax +49 89 289 25310,
`wimmel@informatik.tu-muenchen.de`

Abstract. Firewalls protect hosts in a corporate network from attacks. Together with the surrounding network infrastructure, they form a complex system, the security of which relies crucially on the correctness of the firewalls. We propose a method for specification-based testing of firewalls. It enables to formally model the firewalls and the surrounding network and to mechanically derive test-cases checking the firewalls for vulnerabilities. We use a general CASE-tool which makes our method flexible and easy to use.

1 Introduction

The increasing connection of businesses and other organisations to the Internet poses significant risks: Attackers from the Internet may exploit vulnerabilities in the internal hosts connected to the Internet to gain unauthorised access to the corporate network. Due to the complexity of computer systems, it is impossible to protect an internal host just by making sure that it has no vulnerabilities.

This motivates the use of firewalls [4] to protect a network from the Internet, or subnetworks from each other. Incoming and outgoing traffic is filtered and possibly dangerous services are blocked. Firewalls are complex systems composed of several hard- and software components the correct design of which is difficult, in particular for networks that use more than one firewall (e. g. larger companies). Here, the interplay between the firewalls could introduce vulnerabilities. Absolute correctness of the firewall design and implementation is vital since a single weakness allowing unauthorised access makes it fail. However, testing firewalls is usually confined to applying simple check lists (e. g. [5]), possibly using specialised tools (such as [6]); the reliability of the process depends on the skill of the person in charge.

We propose an alternative approach: we formally model a firewall system, and derive test sequences automatically from the formal specification — following the approach to specification-based testing of [17,18,13]. Testing the firewall with

* This work was partially supported by the Studienstiftung des deutschen Volkes, and by the German Ministry of Economics within the FairPay project.

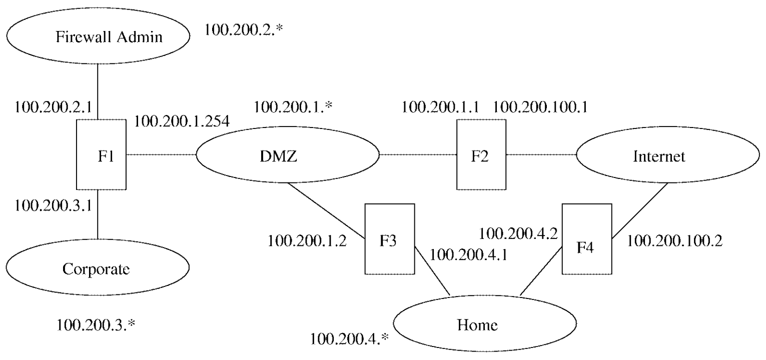


Fig. 1. The Network

these test sequences provides more confidence that the firewall implementation actually provides the desired protection, than ad-hoc testing, especially since the test-sequences are derived with respect to the actual network topology. Our approach is embedded in an easy-to-use CASE framework [9]. Because of its generality, there are few restrictions on the model: Firewall rules need not be of a special form, *stateful* firewalls can be modelled etc. The network model is also flexible, allowing to model possible faults or Trojan horses (malicious code injected by attackers) at the hosts. Various scenarios, such as stress test, spoofing (source address forging), and policy violations, can be tested. Additionally, one can check the firewall specification with a model-checker.

In the next section, we explain our approach using an example network. We then point to related work and conclude.

2 Testing Firewalls

In our approach, we give a (possibly partial) description of a network behaviour that presents a potential threat. From this, a test-sequence is derived automatically which indicates how the system should react to this threat according to the specification. This test-sequence can then be used for actual testing of the firewall.

A further advantage of specification-based testing is that, given a sufficiently detailed specification, one can also determine which values internal variables need to have at certain points of the execution and can use this for more detailed debugging.

2.1 Example Network

We consider an example (in the following called the Network) similar to the one given in [3] (see Fig. 1).

Each interface has its own IP-address. The DMZ (*demilitarised zone*) contains a web-server, a DNS-server and a mail-server.

Here we consider IP-based packet-filtering firewalls. The firewalls should implement the *rule-bases* that are specified below. Each rule specifies a *source*, a *destination*, a *service-group*, the direction of the packet, and the *action* to be taken. The source and destination are host-groups (sets of IP addresses), the service-group specifies a set of services (such as http/https, smtp (e-mail), dns etc.), and the actions we consider here are to *drop* the packets of the corresponding session, or to let them *pass* (for space limitations we do not consider other actions that may be possible, such as writing a log). The service corresponding to the packet can generally be inferred from the number of the source or destination port given in the packet. Here we can only give a few examples of such rules:

– let packets pass only if the direction of the packet complies with the topology of the network wrt. its source and destination (what this means should be obvious in our simple example, e. g. F3 should let a packet out to the Home subnet via the connection 100.200.4.1 only if its source is in the Firewall Admin, the DMZ or the Corporate subnets and the destination is in the Home subnet, F4 should let packets from the Internet into the Home network only if the source address is in the Internet and the destination address in the Home network, etc.)
– let packets with source in the Corporate subnet and destination in the Internet through
– let packets between the mail-server and the Internet, or the Corporate or Home subnets pass if the service is smtp
– let packets between the DNS-server and the Internet, or the Corporate or Home subnets pass if the service is dns
– let packets from the Internet, or the Corporate or Home subnets to the web-server pass if the service is http or https

For example, a packet with source address in the DMZ and destination address in the Internet will be dropped by F3 (by the first rule, since its destination address is not in the Home subnet) and can only go directly via F2 to the Internet (and similarly for the reverse direction).

2.2 Formal Model

We modelled the network containing the firewalls with help of the CASE tool AutoFocus/Quest. AutoFocus[9,10] is a tool for graphically specifying distributed systems. It is based on the formal method Focus, therefore the models have a simple and clear formally defined semantics. AutoFocus supports different views on the system model, describing structure, data types, behaviour and interactions. These views are related to UML-RT diagrams. In addition to modelling, AutoFocus offers simulation, code generation, test sequence generation and formal verification of the modelled systems.

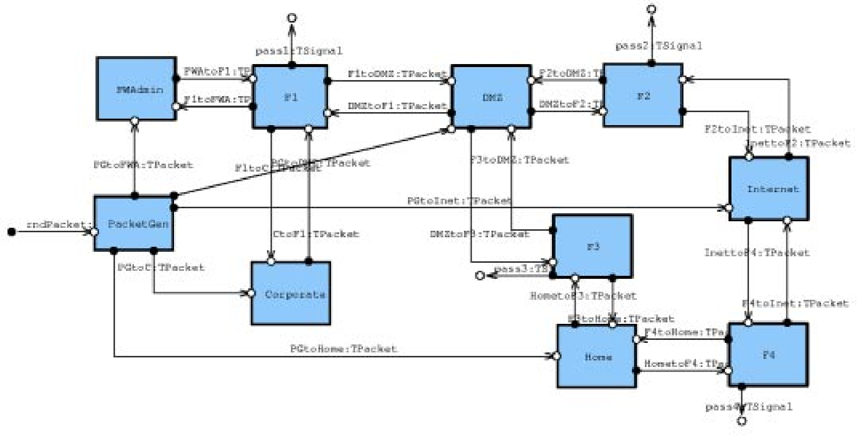


Fig. 2. SSD for firewall system

The **structural view** on our example network system is depicted in Figure 2, as an AutoFocus system structure diagram (SSD). Each network component (subnets and firewalls) is an AutoFocus system component, drawn as a rectangle. These components can exchange data via named channels, which connect output and input ports (filled and empty circles). In addition, there is a special component `PacketGenerator` that produces a random packet and sends it to one of the other components, so that it can start travelling through the network.

The **data type definition** in the model describes network packets, and a function for a consistency condition on packets. As other data types in AutoFocus, they are defined in a functional style, as follows:

```
data TAddress = FWAdmin | Corporate | DMZ | Internet | Home |
                F1 | F2 | F3 | F4 | Mail | DnsSrv | WWW;
data TService = Http | Https | Smtp | Dns | Ping | Ssh;
data TPacket  = Packet(source: TAddress, dest: TAddress,
                service: TService);
data TSignal  = Present;
fun srvKons(Mail,Smtp) = True   |  srvKons(DnsSrv,Dns) = True
  | srvKons(WWW,Http)  = True   |  srvKons(WWW,Https)  = True;
```

Finally, each component in the network model is assigned a specified behaviour, using **state transition diagrams** (STDs). STDs correspond to extended finite state machines (meaning they can have a data state as well as a control state, and communicate with the other state machines). The state transition diagrams for the packet generator and the subnets are straightforward — in the subnets of our current network model, the packets are just relayed at random to other output ports. However, the model is very flexible in this respect, so the functionality could also be changed such that the packets may be manipulated.

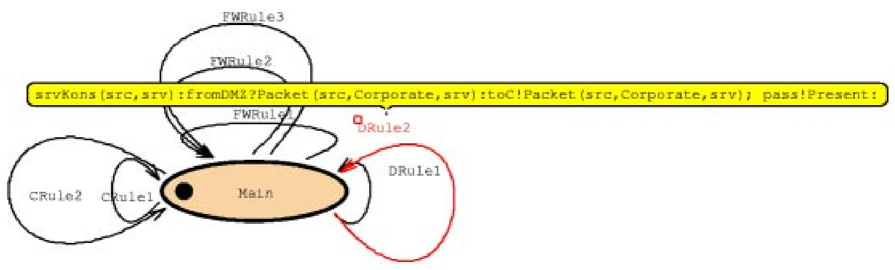

FWRule3
FWRule2
srvKons(src,srv):fromDMZ?Packet(src,Corporate,srv):toC!Packet(src,Corporate,srv); pass!Present:
FWRule1
DRule2
DRule1
CRule2
CRule1
Main

Fig. 3. STD for firewall F1

For the behavioural model of the firewalls, as exemplified by the firewall F1, see Figure 3. Corresponding to each input port of the firewall (which models one of the interfaces of the firewall), there are a number of state transitions, corresponding to forwarding rules of the firewall.[1]As an example, the highlighted transition `DRule2` in Figure 3 can fire if a packet arrives at the interface `fromDMZ` connected to `DMZ`, with destination address `Corporate`, and with a service consistent with its source address. The correspondence is modelled by the function `srvKons` defined above. In case the transition fires, the packet is forwarded to the port `toC`, and the signal `Present` at the output port `pass` gives an indication that it was passed. The other transitions model other forwarding/dropping rules in an analogous way (`FWRule1,2,3` for packets arriving from the `FWAdmin` subnet and `CRule1,2` for packets arriving from the `Corporate` subnet).

AutoFocus is based on a time-synchronous communication scheme, so an execution consists of a sequence of clock cycles.[2] Thus, the forwarded packet can be read by the connected components in the following clock cycle. In all, the highlighted rule corresponds to part of the firewall behaviour as specified in the previous section: packets from the Mail-, DNS-, or Web-Server to the `Corporate` subnet are only passed if the corresponding service entry is correct.

2.3 Testing

In our approach for firewall testing, we use the formal AutoFocus model as a specification to generate test cases. We call this *specification-based test sequence generation* (see e.g. [18]). For this purpose, first test case specifications based on the system model have to be formulated. Test case specifications would be, for example, that we look for executions where a packet arrives at a certain interface of a component, or executions where a packet is dropped by a firewall. These are translated into logic and solved. The solutions are all test cases of a given

[1] Note that often rules in firewalls are presented as sequences, and not as sets like here, but this makes no difference provided the rules are consistent.

[2] Note that this is not a restriction (even though we cannot assume a global clock in a network) because the actions specified here do not depend on global timing.

maximum length satisfying the test case specification. These test cases represent concrete system executions (which exact packets with which data originate at which component, the way they travel etc.), can be depicted as message sequence charts and fed into the actual implementation of the firewall system for testing.

[18] describes a test sequence generation approach based on a propositional solver (SATO), whereas in [13], a variant (also for AutoFocus) is presented that is based on constraint logic programming (CLP). We used the CLP variant for our purposes, as the CLP-solver proved to be more flexible with respect to specification and generation of the solutions.

Thus, with our approach we can systematically generate many (or even all) test cases of a given maximum length to verify chosen security aspects of the firewall implementation. This leads to an improved reliability of the system resulting from the test, as opposed to ad-hoc testing.

In general, our approach supports all kinds of test scenarios that can be specified based on the execution history of the system. Important test scenarios for threats against the firewall example system we tested include the following:

– **Stress test.** For a chosen firewall component, generate all test cases from the specification, where this firewall drops an arriving packet. Thus, the firewall system to be tested can systematically be flooded with packets it should drop.
  As an example, for the firewall component `F1` we have to specify that a packet arrives at one of the input ports, but no `pass` indication is given in the next execution step. See Figure 4 for a corresponding MSC. This MSC shows the route of a packet coming from the Internet that arrives at the DMZ with destination address being the Mail server (so it is correctly passed on by F2), but is then incorrectly relayed to F1 inside the DMZ — for example by a Trojan horse.
– **Spoofing.** In this scenario, packets with forged source addresses are exposed to the system. For example, an attacker on the Internet may try to send packets to the Network that appear to have originated from internal hosts in the Network in order to defeat security mechanisms (such as the inner fire-walls) that rely on the source information of packets. These test cases can be generated in a similar way as above, by specifying that the packets generated by the `PacketGen` component should have forged source addresses. Spoofing is described in more detail in [2].
– **Policy violations.** As explained in [8], firewall systems have to be based on a security policy. In [8], this policy is given in the form that, if a packet *was in* a certain subnet, and *reaches* another subnet, a certain condition on its source and destination address and service has to be fulfilled. These policy statements can also be used for test sequence generation. Two variants are possible — first test case specifications which fulfill the policy rules and thus check if the firewall system correctly lets packets pass, and second test case specifications that violate the policy. The latter can be derived by negating the condition on the packet data and lead to test cases that check if the firewall correctly drops packets. The execution in Fig. 4 actually represents

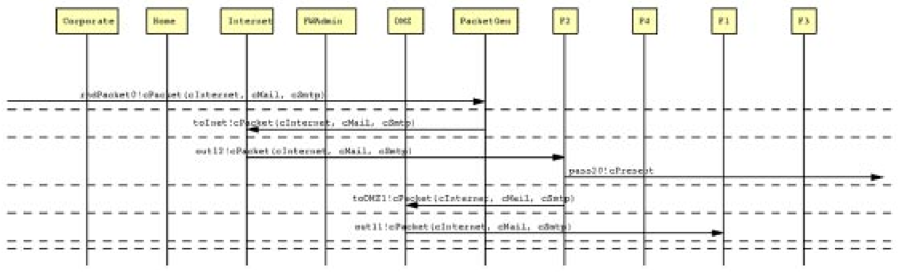

Corporate
Home
Internet
FWAdmin
DMZ
PacketGen
F2
F4
F1
F3
sndPacket0!cPacket(cInternet, cMail, cSmtp)
toInet!cPacket(cInternet, cMail, cSmtp)
out12!cPacket(cInternet, cMail, cSmtp)
pass20!cPresent
toDMZ1!cPacket(cInternet, cMail, cSmtp)
out11!cPacket(cInternet, cMail, cSmtp)

Fig. 4. Test Case for firewall FW1

both a test case for fulfilment of the policy given in section 2.1 (`smtp` packets from the Internet to the mail server can be passed to the DMZ) and a violation (those packets must not be relayed on to the `Corporate` or `FWAdmin`) subnets.

Systematic Selection of Test Sequences. For a given test case specification, the number of test cases can get fairly large — especially with more complex data types than in our example. In our case, we can use domain-specific knowledge to improve coverage. Firstly, the maximum length of the test sequences can be restricted to the diameter of the network, provided that only one packet at a time is considered and the components do not maintain states. In addition, we can separately generate a number of test cases with fixed packet origin, source address, destination address or service specified in addition to the original test case specification, to ensure these possibilities are tested. In a further stage, this is also possible for certain combinations of these parameters (e.g., tests where packets originating from the Internet with service "Smtp" are involved etc.).

Performance. Computing test cases of the above kind for our example network takes about .1s per test case, measured on a SUN UltraSparc with 1GB of main memory running at 400MHz.

3 Related Work

[7,8] introduces a language for expressing global network access control policies and algorithms to compute filters for such policies and to check filters against policy violations. [3,14] present a firewall management toolkit. Contrary to ours, the intention of this work is not firewall testing, but managing the configuration, and it is assumed that all filtering devices work properly [14, p.2]. [15] uses model-checking to analyse network vulnerabilities.

A Petri-net model of firewalls is given in [16]. References to work on specification-based testing are in [18]. The Focus method underlying AutoFocus has previously been used for systems security, e. g. in [12,11,1].

4 Conclusion and Future Work

We proposed a method for specification-based testing of firewalls, enabling one to formally model the firewall and the surrounding network and to mechanically derive test-cases checking the firewall for vulnerabilities. We used a general CASE-tool which makes our method flexible and easy to use. We demonstrated our approach with an example firewall.

In future work we will consider advanced network and firewall designs, such as authentication headers (using cryptography), virtual private networks, and distributed firewalls. It would be desirable to have a higher-level language for the security policies that is automatically translated into rules (following [8]).

Also, our approach opens up the possibility to go beyond test-sequence generation and perform the actual testing automatically, on an actual firewall system.

Acknowledgement. Helpful comments by Joshua D. Guttman on a draft are gratefully acknowledged. Many thanks also go to Alexander Pretschner for helpful suggestions on constraint-based test sequence generation and comments on this paper, and to Heiko Lötzbeyer for the implementation work.

References

1. M. Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *Theoretical Aspects of Computer Software (TACS '01)*, LNCS. Springer, 2001.
2. S. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, 1989.
3. Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Security and Privacy*, 1999.
4. W. Cheswick and S. Bellovin. *Firewalls and Internet Security: repelling the wily hacker.* Addison-Wesley, 1994.
5. UK IT Security Evaluation and Certification Scheme. UK ITSEC Certification Report No. P117 – CyberGuard Firewall for UnixWare, 1999.
6. M. Freiss. *Protecting Networks with SATAN.* O'Reilly, 1998.
7. J. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, 1997.
8. J. Guttman. Security goals: Packet trajectories and strand spaces. In R. Gorrieri and R. Focardi, editors, *Foundations of Security Analysis and Design*, LNCS. Springer, 2001. Forthcoming.
9. F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.
10. F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights – An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

11. Jan Jürjens. Composability of secrecy. In *International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS 2001)*, volume 2052 of *LNCS*, pages 28–38. Springer, 2001.
12. Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (International Symposium)*, volume 2021 of *LNCS*, pages 135–152. Springer, 2001.
13. H. Lötzbeyer and A. Pretschner. Testing concurrent reactive systems with constraint logic programming. In *2nd Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.
14. A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, 2000.
15. R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2000.
16. C. Schuba. *On the Modeling, Design, and Implementation of Firewall Technology.* PhD thesis, CERIAS, Purdue, 1997.
17. G. Wimmel. Specification Based Determination of Test Sequences in Embedded Systems. Master's thesis, Technische Universität München, 2000.
18. G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10, 2000.

Academic vs. Industrial Software Engineering: Closing the Gap

Andrey N. Terekhov[1] and Len Erlikh[2]

[1] St.-Petersburg State University, LANIT-TERCOM
Bibliotechnaya sq., 2, office 3386
198504, St.-Petersburg, Russia
ant@tercom.ru
[2] Relativity Technologies, Inc.
1001 Winstead Drive
27513, Cary, NC, USA
Len.Erlikh@relativity.com

Abstract. We argue that there is a gap between software engineering cultivated in the universities and industrial software development. We believe that it is possible to get academia and industry closer by starting projects that will require solution of non-trivial scientific tasks from one side and long-term commitment to create a product out of this research solutions from the other side. We illustrate our position on a real-world example of collaboration between an American company Relativity Technologies and research teams from St.Petersburg and Novosibirk State Universities. We also point out that the current economic situation in Russia presents unique opportunity for international projects.

Introduction

Industrial programming is usually associated with big teams of programmers, strict timelines and established solutions and technologies. On the other hand, the main goal of academic research is to find new solutions and break existing stereotypes. Unfortunately, amazingly low percentage of scientific results makes their way into practice, and even when they do, the process is very slow.

In the meantime, practice always required a solution of the tasks that are infeasible from the point of view of the existing theory. Today this common truth takes on special significance for software engineering because the number of its applications really exploded. A special emphasis on this problem was made by academician A.P. Ershov. By the way, it is little known that for several years A.P. Ershov worked as a consultant in research institute "Zvezda" of LNPO "Krasnaya Zarya" (Leningrad). By tradition of those times this job was not paraded, because the institute worked in the area of government communications, so later on even specialists who knew A.P. Ershov personally, were surprised that a scientist so famous was spending his time on such "utilitarian" problems.

It is a well-known situation when practitioner is posing a problem and theoretician is reasoning why this task is unlikely to be solved. But the proof of impossibility of correct solution of the problem does not satisfy the demand for it, so practitioners start to seek partial or heuristic solutions or try to use "brute force" method.

In this article we try to show that even on this shaky ground it is better to use specialists that know the theoretical restrictions, complexity estimations for this or that solutions, optimization methods and other traditionally scientific knowledge. This sounds pretty obvious, but somehow the chasm between academic and scientific communities is very difficult to close. What are the main reasons for this?

It is well-known that software engineering is differing from pure mathematics or even computer science. Proved theorem or complexity estimation for some algorithm are results by themselves, and there are no other requirements for their creation other than scientists' talent, pen and paper. On the other hand, in software engineering a new interesting approach or even working prototype does not guarantee that they will lead to the successful and ready-to-use product. To achieve this, one should add up large teams, investments and strict industrial discipline. In this respect, software engineering is close to elementary-particles physics or physics of ultralow temperature etc.

Nevertheless, there are some positive examples and we believe that they could be considered as role-models for promoting collaboration between industry and science. We try to illustrate this process on the example of creating an automated reengineering tool RescueWare, which automates reengineering of legacy software, i.e., conversion of systems written in COBOL, CICS, embedded SQL, BMS, PL/I, ADABAS Natural and other languages, working mostly on IBM mainframes to C++, VB or Java. Software reengineering does not end up in simple translation from one language to another — completlely different schemes of dialog with the user, access to legacy databases, recovery of lost knowledge about the program make this task much more difficult.

This project was carried out by large international team, which was geographically spread from North Carolina (USA) to St.Petersburg and Novosibirsk. The customer for this project was an American company Relativity Technologies, and the team that worked on this project included scientists from St.-Petersburg and Novosibirsk universities, LANIT-TERCOM company and A. P. Ershov Institute of Informatics Systems of Siberian Branch of Russian Academy of Sciences. The total investment in this system amounts to more than 400 man-years.

Our collaboration began in 1991, and during these years we have overcome a lot of difficulties, mostly related to cultural difference and lack of understanding. Finally, as a result of common efforts we created a science intensive product RescueWare Workbench, which was recognized by Gartner Group as a best product in 2000 in the area of legacy understanding. In the following sections we give a detailed descriptions of the hard problems that we had to solve in order to make this project a success.

1 Architecture for Multi-language Support

From the very beginning we were oriented on creation of *multi-language* translator, so one of our first tasks was to design a unified intermediate language (IL) for our system. The idea is that program transformation is two-staged: at first the program is converted to IL, and then to the target language. When there are M input and N output languages, this approach makes it possible to limit the amount of work to $M + N$ compilers instead of $M * N$.

Under the name of UNCOL this approach is known for more than 30 years. A lot of teams were trying to implement it, but only a few managed to create something tangible [1,2]. Now what is the problem? From the theoretical point of view, all langugages are computationally equivalent, and thus conversion should be quite simple. The difficulties arise when we are measuring the quality of output program not only from the point of view of its performance, but mostly from the point of view of naturality of program's structure in this or that language.

Every programming language gives us some means of expression and a discipline of using them. At that, the most quality from the point of view of quality and easiness of maintenace could be attributed to those programs that meet the requirements of this discipline.

The problem appears when notion of “natural program” in one language contradicts with the same notion in other language. For instance, using GOTO statements is usual for COBOL, but in Java there are no such statements at all. Some special and sometimes non-trivial transformations, dependent both on initial and target platforms, may be required to solve this problem.

Let us name the main levels of program representation:

1. Control flow representation
2. Data flow representation
3. Representation of values

Thus IL must contain abstract means for program representation at all these levels, and the transformation will look as follows: first language constructs of the source language are “raised” to abstract intermediate representation and then they are “lowered” to concrete representation in the target language. The degree of abstraction should be carefully chosen to make sure that this lowering down leads to natural projections to the target language.

The most important condition for the proposed approach is the *orthogonality* of translations to IL and from it. This means that the representation of source program in IL should not depend on the target language.

Finally, IL should be extendable, i.e., it should contain features that permit to build new constructs without changes to all existing passes of compiler.

The weak point during IL design is the choice of data types and standard opertions. While the set of control constructs in different languages is more or less suitable for unification, data types system could be significantly different. It is inexpedient to simply combine all types of the source languages, because addition of new language will require major changes to existing compiler functionality.

To solve this problem, it is important to add not only standard data types to IL, but also add formalism of higher order — *type constructor* — which could be regarded as a function, which produces a new type out of several given types. For example, abstractions such as structures, arrays and pointers could be considered as type constructors. Indeed, the structure could be defined as a type constructor, which receives a set of field types and creates a structured types with the fields of corresponding types.

Finally, usage of type constructors eases runtime support, for we can consider type constructor, which is treated as an abstract dynamic data type and could be used only through runtime support functions. This ensures that the addition of a new entity requires changes in only one compiler pass — namely, of the pass, which creates this entity. After that all handling of this entity will be transparent to the compiler and conducted through conversion of type constructors.

We believe that this task presents a good example of semantic gap between academic research and industrial programming. The idea of unified IL makes sense only in large-scale projects, and these projects are out of acaedimic scope. On the other hand, average programmer in the industry just does not possess all the knowledge, which is required for successful implementation of this approach.

Note that "naturality" of IL structure, which was mentioned above, is also a good example of difficult to formalize notions. These notions are often necessary to solve usual everyday tasks. Another example of difficult to formalize notions is the definition of "good program" criteria [3].

2 Re-modularization. Class Builder

Another interesting task that we encountered during creation of automated reengineering tool is re-modularization of programs into components [4,5].

This task could be described as follows: there is a large application that consists of multiple files, which contain declaration of data and procedures. Variables and procedures from different files are interacting with each other through some external objects, which we called *dataports*. For legacy systems usual dataports are CICS statements, embedded SQL and other infrastructure elements.

This task was formalized as follows: application was represented as a graph with application objects as junctions of various types (variable, procedure or external object), and relations between them as graph edges. Edges are also typed (for instance, procedure call, variable usage in procedure, working with external object through variable etc.). Also, each edge is attributed with some number, which defines the "power" of this relation. For example, the power for procedure call relation could be defined by the number of parameters passed: the more parameters we have, the more we want to place both callee and caller to one component.

This graph should be divided into some areas of strong connectivity. To do this, we introduce the notion of gravity between two nodes, which is calculated as sum of powers of all edges connecting them multiplied to coefficient defined by the edge type, minus some constant, which is defined by the pair of edge types.

Some negative part of the formula — the repulsive force — should be included, otherwise we will always get one monolith. For instance, it seems natural to add repulsive forces between dataports and any procedures. This way we want to separate all procedures first and only when two procedures are using too many common variables, then the gravity force prevails.

Then by complete enumeration we find those junction sets, for which the sum of gravity force between themselves and the junctions from other sets are maximal (of course, gravity forces with the junctions of other groups are taken with minus sign).

It is clear that this good idea will not work in real life, because the number of graph junctions in real applications is way too much to use exhaustive searches. But we managed to find some heuristic approaches, which made it possible to achieve practical results.

First of all, we fixed some coefficients for different types of edges and repulsive forces for different types of junctions. However, the user can assign coefficients on his own if he believes this to be of importance for his application.

Secondly, in the complete graph of application we will start with sub-graph, which consists of the junctions corresponding to external object plus edges and junctions of any other types, which connect these external objects. The heuristics is that we believe external objects to be cross-linking and thus we add repulsive forces only for them. On the other hand, if two external objects are using a lot of common variables and procedures, then nothing prevents them from ending up in one component.

Thirdly, we will regard all edges of reduced graph as being of the same type, but we will define the power of each edge so that the more relations there are (not only direct, but also transitive ones), the bigger is power. To formalize this notion of "bigger", we will use the physical model.

To calculate the power of edges in reduced graph we proposed to use the model of electric mains. In this model we will consider that there exists a wire between any two junctions of the input graph that have a positive gravity force between them and we will equal this force to the conductivity of the wire (let us remind that conductivity is the reverse function of resistance). Then as a power of edge between two junctions of the reduced graph we can use the complete conductivity of the electrical network between them. The complete conductivity could be calculated using Kirhgof equations, so we need to solve a set of linear equations. The complexity of this task is equal to finding the inverse matrix.

3 Program Slicing

Let us suppose that a legacy system performs ten functions, seven of which are no longer needed, but the remaining three are in active use, and as it often happens with legacy programs, nobody knows *how* these three functions work. In this case it is necessary to create a tool for deep analysis of the old programs, which can help maintenance engineer to find and pick out the required functionality, put the corresponding parts of the program into a separate module and reuse it in the future, for instance, to move it to modern language platform.

The solution of this task is based on creating static slices of the program and their modifications. We regard program slices to be a subset of program statements that presents the same execution context as the whole program. Slice is a program that contains the given statement and some other statements of the initial program, namely those that are related to this statement.

The following methods are implemented in RescueWare for automation of business rule extraction (BRE):

- Computational-based BRE
- Domain-oriented BRE
- Structure-oriented BRE
- Global BRE

All these methods assume generation of syntactically correct independent programs that preserve the semantics of the original code fragments.

Computational-based BRE forms the functional slice of the program, based upon the execution path and data definitions that are required to calculate the values of the given variable in the certain point of the program [9]).

Domain-oriented BRE generates functional slice of the program, which is received by fixing the value of one of the input variables. Being based on theory of program specialization, domain-oriented BRE is best suited to separate calculations with many transactions and mixed input, into a series of "narrowly specialized" business rules with only one transaction for each of them.

Structure-oriented BRE makes it possible to divide the programs written as a single monolith into several independent business rules, based on the physical structure of the initial program. Also, an additional program is generated that calls the extracted slices in a proper sequence and using the correct parameters (ensuring the semantic equivalence of this program to the initial one). This method is best suited to divide old large programs into parts that are easier to handle.

Finally, global BRE helps to apply all of the methods mentioned above to a number of programs simultaneously, and thus supports BRE on system-wide basis.

Notwithstanding all automation, the choice of slicing points in the program and the sequence of application of different BRE methods are left to the human analytic, which decomposes the initial system. There are several natural points to start applying BRE, for instance, the places where the calculated values are stored to the database or printed to screen. Of course, intelligent choice of candidates for business rules requires knowledge about functions of the initial system.

On closer examination it often turns out that the methods used are differing from one module to another. Moreover, it is sometimes useful to apply different BRE methods subsequently to the same program.

4 Conclusion

As of right now, products such as RescueWare are not really typical for the market, because creation of RescueWare required solution of *many* scientifically difficult problems. Let us briefly mention other achievements: relaxed parser that ensures collection of useful information even for quite distant dialects of the language, different variants of data flow analysis, using sophisticated algorithms of pattern matching for identification of structure fields in PL/I etc.

Of course, not all projects require investment of this amount of research. Even in our own practice not all projects both in Russia and USA were so rewarding. Our present understanding of importance of education and training in software engineering came only after a lot of painful experience. The contacts with the universities are important for the industry not only because of the talent pool accumulated there, but also because of new ideas and scientific breakthroughs that can make the sofware product a success. This road is very difficult; sometimes the scientific research contradicts with the strict timelines and discipline, but the potential payoff of this approach is immense.

Finally, we would like to emphasize that Russia is in a special position to make this vision come true, because it has an undoubted advantage in the level of education at the software market. We hope that our experience of successful cooperation of American company with Russian scientists could serve as a good example for many Western companies.

References

1. A.P. Ershov "Design specifications for multi-language programming system", Cybernetics, 1975, No. 4. P. 11–27 (in Russian)
2. GCC home page. http://www.gnu.org/software/gcc/gcc.html
3. I.V. Pottosin "A "Good Program": An Attempt at an Exact Definition of the Term" // Programming and Computer Software, Vol. 23, No. 2, 1997. P. 59–69.
4. A.A. Terekhov "Automated extraction of classes from legacy systems" // In A.N. Terekhov, A.A. Terekhov (eds.) "Automated Software Reengineering", St.-Petersburg, 2000, P. 229–250 (in Russian)
5. S. Jarzabek, P. Knauber "Synergy between Component-based and Generative Approaches", In Proceedings of ESEC/FSE'99, Lecture Notes in Computer Science No. 1687, Springer-Verlag, P. 429–445.
6. M. Weiser "Program Slicing" // IEEE Transactions on Software Engineering. July 1984. N 4. P. 352–357.
7. T. Ball, S. Horwitz "Slicing Programs with Arbitrary Control Flow" // Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. 1993.
8. M.A. Bulyonkov, D.E. Baburin "HyperCode — open system for program visualization" // In A.N. Terekhov, A.A. Terekhov (eds.) "Automated Software Reengineering", St.-Petersburg, 2000. P. 165–183 (in Russian)
9. A.V. Drunin "Automated creation of program components on the basis of legacy programs" // In A.N. Terekhov, A.A. Terekhov (eds.) "Automated Software Reengineering", St.-Petersburg, 2000. P. 184–205 (in Russian)

A Method for Recovery and Maintenance of Software Architecture

Dmitry Koznov, Konstantin Romanovsky, and Alexei Nikitin

St.-Petersburg State University, Faculty of Mathematics and Mechanics
Department of Software Engineering
198504 Bibliotechnaya sq., 2, St.-Petersburg, Russia
{dim,kostet,lex}@tepkom.ru

Abstract. This paper proposes a method for recovery and subsequent maintenance of the architecture for actively evolving software systems. The method's underlying idea has to do with constructing a basic set of the architecture elements, which set would then be used for creating different views of the system. The responsibilities of the modules, which make up the software system, as well as elements of the system's data dictionary, are considered elements of this basic set. Of special meaning is the fact that the system is being actively maintained and developed, that the knowledge about it is accessible, but needs to be alienated from the respective bearing media, to be generalized and formalized.

1 Introduction

While many software products, for which the architecture was never formalized, may exist for years and be successfully maintained, still one can face a situation when formalization of the system architecture becomes a priority task. As a result, a lot of architectural imperfections in the system reveal themselves, and so there comes such moment in the life of the product, when its further development is impossible without improving the architecture, which requires formalizing the latter. Even various methods of analyzing and designing software systems have been spreading widely [11,9,10], it is a greate problem to use such methods in the situation like this. On the one hand this activity can cause serious internal restructuring of the system. On the other, we cannot ignore the commercial aspects of the software development process, issuing of new versions, service packs, and the implementation of new customer's requests.

Thus there is a problem: how to perform reverse engineering of the architecture of the actively evolving system, with most effective gathering and formalisation of the meta-information on the system. In the same time the architecture views should be supported in actual while the system will evolve.There are many methods and tools of reverse engeeniring designed for recovering the knowledge about a system architecture based on source code parsing (Refine/C, Imagix 4D, Rigi, Sniff [6,9], RescueWare). Also there are formal methods of the reverse engineering [3].

All these methods have the same week point: absence of mechanism for synchronization of the model recovered with the source code of the software system. This problem makes very ineffective the use of such methods for actively evolving systems. From this point of view, there are more perspective Use-case driven methods [4], because the models that were derived with this method should be more stable to the system's changes on the practice. Data-mining methods [7], relying on the inner links between system elements are also interesting from the same viewpoint. However, all these methods are more suitable for the architecture of the "dead" system.

Round-Trip development methods, that are provided with some CASE-packages (e.g. Rational Rose, Together), enables the bi-directional connection of source codes and the architecture view. But the quality of the information visualized is unsatisfactory, because in fact, we do not get any new meta-information on the system, but only visualize the source code structure.

2 Starting Point

Let's formulate the basic problems that are to be solved with the presented method of architecture recovery:

1. The utilization of the features of the "living" system for the most effective architecture recovery;
2. The ability of the maintenance and further development of the architecture view of the software system;
3. The ability of step-by-step adopting of the process of development and support of the architecture without any serious damage to industrial requirements to the process.

3 The Method

3.1 Structure of the Model

The system architecture model proposed with this method consists of the following views: Structure views, Dynamic views and Physical view.

The basis of describing the system architecture is a System Structure description, which it is proposed to implement on the physical level (Physical view) and the logical level (Structure views). The Dynamic views are needed in order to determine the main scenarios of the system's operation.

The necessity of different kinds of views of the software is well known [1,5,8]. With the method in discussion, it is proposed to construct also a set of various Structure and Dynamic views, which is motivated by the following considerations:

1. The participants of the project, whose positions are on different levels of the hierarchy (managers of different levels) need information about the system to be specially adapted;

2. There exist both a vertical division of the system (by business functions) and a horizontal one (by tiers – e. g., User Interface, Business-Logic, Data Access);
3. There are different modes of packaging and deployment of the system;
4. In order to compile the entire project, information about the structure of storing the source codes of the system is needed;
5. Organizational structure of the enterprise affects decomposition of the system.

Structure Views. It is proposed to organize Structure views in the form of UML class diagrams. We herewith associate some part of the system (subsystem) with a class. The subsystems are organized into a multiple containment hierarchy, with the only restriction that the aggregated subsystems become invisible from the context, in which their aggregate exists. The different views should be constructed upon the same basic set of subsystems, but in other respects do not require any special matching. Associations between the subsystems, which reflect their semantic connections, are possible on each level.

Dynamic Views. With the help of the dynamic views it is proposed to represent the main scenarios of the system's operation. One can associate with each level of a structure view a dynamic view, which would explain how the subsystems interact with each other. For this, it is proposed to use the UML Collaboration Diagrams.

Physical View. This view is designed for inventory of the software source codes and for associating them with elements of the structure and dynamic views. In the view, the following items are considered:

1. Set of program modules (e. g., for Microsoft Visual C++ these are projects);
2. System data dictionary, which consists of:
   a) Persistent data (logical data of the system and the corresponding physical media);
   b) Channel data, with which subsystems exchange not through persistent-structures (physically media are absent, only logical elements of the data are there);
   c) Configuration data that constitute a kind of persistent data, but are responsible for tuning the system algorithms (logical data and the corresponding physical. media).

3.2 Constructing the Basic Set

The foundation of this method consists in constructing a basic set of subsystems, from which different structure views will be built. In order to keep the correspondence of the views to each other and to make the maintenance easier, all elements of such views are to be subsets of the basic set of subsystems. So, the views themselves are hierarchical coverages of the basic set: the elements of each view form an aggregation hierarchy and the set of leaf nodes in that hierarchy is a usual coverage of the basic set. The basic set is constructed from responsibilities, which are assigned to the system modules (3–5 responsibilities per each module) and with elements of the system data dictionary.

Elements of the basic set of subsystems may be included in subsystems of different views; therefore, for them multiple containment is permissible.

When construction of the basic set is finished, all participants of the development process may build for themselves a package of structure and dynamic views that they need. A coordination of different views is provided by due to integrity of the set of basic elements, which are placed on different diagrams.

4 Conclusions and Further Research

The method presented is designed for recovery, formalization and further maintenance of architecture of the actively evolving software systems.

At the moment there are some open problems to focus the investigation on. These problems are mainly about the adoption of the method presented in the industrial process of software development:

1. Clear mapping of the presented method's notions to UML;
2. Organization procedure of the adoption of the method presented.

References

1. J.Rumbaugh, I.Jacobson, G.Booch. The Unified Modeling Language Reference Manual. — Addison-Wesley, 1999.
2. Daniel Aebi: Data Re-Engineering — A Case Study. — ADBIS 1997, 305–310
3. Liu, H. Yang H. Zedan: Formal Methods for the Re-Engineering of Computing Systems: A Comparison // X. COMPSAC '97 – 21st International Computer Software and Applications Conference, – 1997
4. Christian Lindig, Gregor Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. — ICSE 1997, 349–359
5. Philippe Kruchten : 4+1 view model of architecture. – IEEE Software, November 1995, 42–50
6. Berndt Bellay and Harald Gall : A Comparison of four Reverse Engineering Tools // Working Conference on Reverse Engineering (WCRE '97) — October 6–8, 1997
7. K. Sartipi, K. Kontogiannis, F. Mavaddat : Architectural Design Recovery using Data Mining Techniques. // IEEE European Conference on Software Maintenance and Reengineering (CSMR 2000), pages 129-139, 29 Feb.–3 March 2000, — Zurich, Switzerland, 2000
8. Philippe Kruchten: The Rational Unified Process. — Addison Wesley Longman, Inc, 1999
9. M.N. Armstrong, C. Trudeau: Evaluating Architectural Extractors. — IEEE Software 1998, 30–39
10. Ivar Jackobson: Object-Oriented Software Engineering. — Addison-Wesley, 1993
11. Grady Booch: Object-Oriented Analysis and Design. — The Benjamin/Cummings Publishing Company Inc., 1994

An Empirical Study of Retargetable Compilers

Dmitry Boulytchev and Dmitry Lomov

St.-Petersburg State University, Faculty of Mathematics and Mechanics
Department of System Programming
198504, Russia, St.-Petersburg, Bibliotechnaya sq., 2
Tel./Fax: +7(812)428-71-09
{db,dsl}@tepkom.ru

Abstract. The paper describes evaluation results of some modern retargetable codegeneration frameworks. The evaluation was performed to estimate applicability of these approaches in hardware-software codesign domain so ease of retargetability and efficiency of generated code were main criteria. Evaluated tools were selected from National Compiler Infrastructure (NCI) project.

1 Introduction

Hardware-software codesign is modern technique aimed to obtain high productivity of real-time and embedded systems. Key feature of this approach is simultaneous development of the program and the target processor or specialization of parameterized processor architecture to match target software application.

Generally, codesign implies iterative development. Each iteration consists of building new hardware description based on previous profiling and efficiency estimations, building (somehow) compiler, debugger, simulator, compiling and possible debugging target application, profiling and estimation of profit/loss. So building set of retargetable tools is basic and very frequent procedure.

Despite a number of retargetability techniques building of compiler still remains matter of art. Since main codegeneration approaches are investigated well the contiguous tasks (supporting of calling and linking conventions, building debugger and profiler etc.) should be solved (semi)–manually. The most crucial problem of building machine–dependent code optimizer also remains open.

Here we describe most recent retargetable codegeneration frameworks that look most preferable for purposes under considerations and briefly present the results of their evaluation (see [4] for details).

2 Retargetability Issues

Compiler's retargetability is usually understood as its ability to be re–targeted to another machine platform "automatically" or "nearly automatically". This implies building of codegenerator from some description. Ideally such a description should be extracted from description of actual hardware but as for now there is well-known semantic gap between hardware description and codegenerator description. So now transition from hardware to codegenerator is mainly proceeds as follows: first verbal instruction set description is produced, then codegenerator description is written from it.

Starting from the most fundamental results in code generation area [1,3] main retargetability technique stays tree pattern matching and dynamic programming. A number of ways to exploit this idea are investigated [6,7,13,24,25]; also there are a number of compilers based on them. These methods often considered as means of *instruction selection* so register allocation and instruction scheduling should be done separately.

Similar attribute-grammar based method described in [14]. Most of heuristic codegenerators use this notion.

Quite different approach suitable for VLIW processors codegeneration is suggested in [15,20]. This approach is based on covering of so–called *split–node DAG* that reflects possibilities of parallel execution of DAG nodes with primitive instructions — so instruction selection, resigter allocation and scheduling are all performed simultaneously. To provide feasible schedule *binate covering* method is used [17,20]. Unfortunally there is no compiler built on this technology so there is nothing to evaluate yet.

Finally there are some novel approaches to retargetable codegeneration including automatic building of codegenerator from architecture or instruction set description [19,23,31]. However tools presented there are either far from real industrial compilers or not accessible for evaluation.

3 Criteria and Methods

The basic factors to be taken into account are, of course, quality of generated code and ease of retargetability.

To assess quality of generated code, we compare the performance of several benchmarks on architectures that the tools being evaluated are already ported. We use Intel Pentium III and Sun SPARC processors for this purpose.

We used benchmarks developed by Standard Performance Evaluation Corporation (SPEC)[1]. This is an industry-standard set of benchmarks to assess quality of computer systems. However SPEC was not initially designed to be used as tool for compilers' performance comparison. For example it contains benchmarks written in different programming languages (Fortran, C++, C) and moreover utilized some specific compiler-dependent features. So we changed some SPEC benchmarks to make them appropriate for other compilers being evaluated.

Then it turned out that some of compilers were unable to compile some SPEC tests correctly either at whole or with some optimizations turned on. So we provide some auxilliary narrow set of benchmarks beyond basic SPEC set. These benchmarks are:

[1] `http://www.spec.org`

- *bzip2*: BWT-based data compression utility, by Julian Seward
- *gzip*: LZW-based data compressor, by Jean-Loup Gailly
- *ranking*: Implementation of Symbol Ranking text compression algorithm, by Dmitry Lomov

All of these benchmarks were compiled by all of evaluated tools with major optimizations turned on.

In according to reasons mentioned above we evaluated all tools in according with measures listed below:

- *Soundness*: describes how close evaluated tool is to real industry compiler. We express soundness in percents of all passed SPEC benchmarks.
- *Selected Performance*: describes peak compiler performance. To evaluate selected performance we compared compilers on narrow set of benchmarks. We express selected performance using formula $K/absolute\ running\ time$, where K is some specially selected constant.
- *Overall Performance*: describes performance evaluated on full SPEC suite. In addition we use non-retargetable platform-native compiler for comparison purposes. Overall performance expressed in percents of best performance among all tools

Informally speaking, selected performance reflects some expectations about compiler's performance after all bugs eliminated. Note that this estimation is rather optimistic because fast code can probably be generated due to inaccurate analysis during optimizations.

To assess ease of retargetability, each tool evaluated has been ported to a "toy" instruction set, designed for a specific algorithm. Symbol Ranking was choosen as target algorithm. This estimation is also optimistic because it is much simpler to port compiler for special fixed application.

4 Evaluated Tools

We selected compilers from *National Compiler Infrastructure (NCI)*[2] project. The project was started under support of DARPA and NSF by major USA Universities (Harvard, Princeton, Stanford, Rice etc.)

On the other hand we have chosen legendary `gcc` compiler [30] as most authoritative industrial optimizing C compiler.

NCI project is aimed at developing interoperable framework for constructing retargetable, optimizing compilers. Combination of these two qualities – *retargetability* and *optimization* – is crucial for hardware-software codesign. Without good retargetability, co-design cycle becomes unbearably long; without optimization, the whole idea of co-design is compromised, as non-optimizing compiler does not employ features of the target architecture to its best. NCI project compilers represent current state-of-the-art in developing easily retargetable, optimizing compilers.

[2] http://www.cs.virginia.edu/nci/

Currently three C compilers are available from NCI: `SUIF/MachSUIF`, `lcc` and `VPO`-based compiler. We evaluated all of them.

SUIF and MachSUIF. `SUIF` *(Stanford University Intermediate Format)* [18] and `MachSUIF` *(Machine SUIF)* [29] are developed in Stanford and Harvard Universities correspondingly. Both systems are parts of NCI project. Unfortunately SUIF/MachSUIF compiler is not ported to Sun SPARC so it is not evaluated at that platform.

VPO-based compiler. `VPO` *(Very Portable Optimizer)* is a part of Zephyr[3] project. The project is in turn part of NCI.

lcc compiler. `lcc` compiler was developed in Princeton University, USA, since 1991 and later was also involved into NCI project [9,10,11,12].

5 Results and Conclusions

Unfortunately at the time of writing the paper only selected performance evaluation was completed on Sun SPARC platform. The result of the evaluation is shown at figure 1. The other results are to appear at
http://oops.tepkom.ru/eval.html in near future.

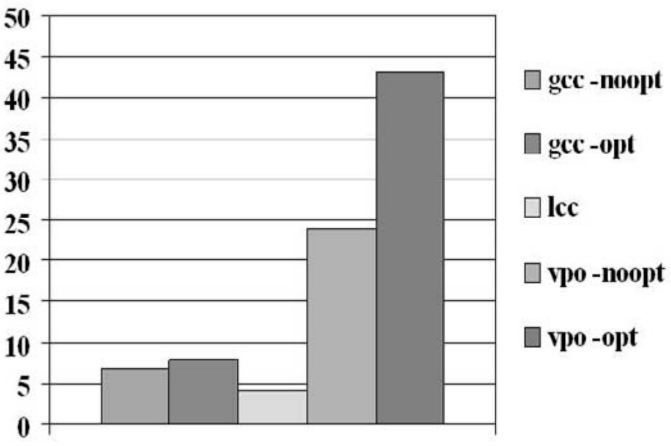


Fig. 1. Selected performance on Sun SPARC, 1000/absolute time

Results of soudness evaluation on Intel Pentium III platform are shown at figure 2. We can conclude that neither `SUIF` nor `VPO` turned out to be ready-to-use compilers — during the evaluation we encountered lots of bugs that had to be fixed.

[3] http://www.cs.virginia.edu/zephyr

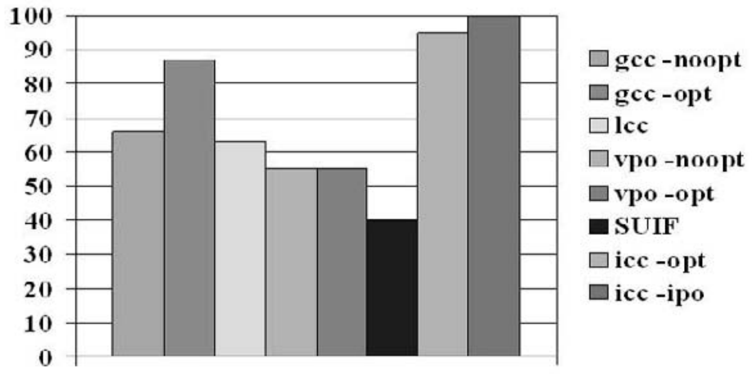


Fig. 2. Soundness on Intel Pentium III, % of passed benchmarks

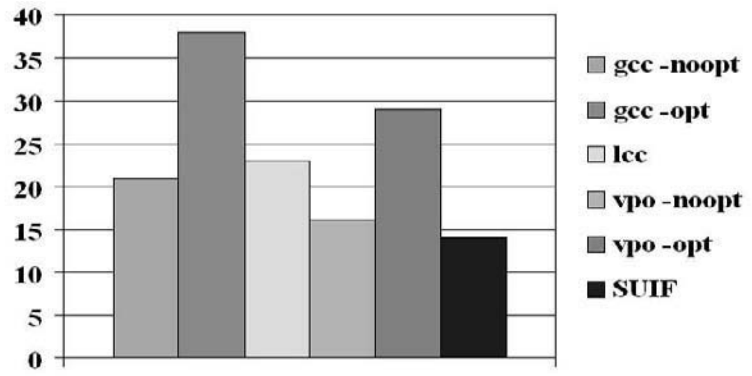


Fig. 3. Selected Performance on Intel Pentium III, 1000/absolute time

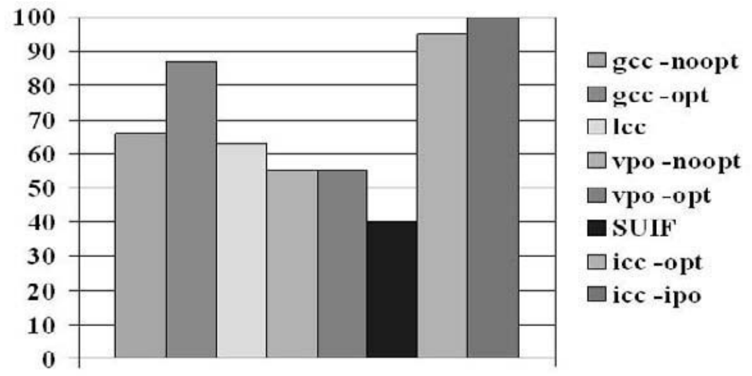


Fig. 4. Overall Performance on Intel Pentium III, % of best performance

Selected and overall performance evaluation results at Intel Pentuim III are shown at figure 3 and figure 4, correspondingly. We have choosen Intel C/C++ compiler (`icc`) as non-retargetable platform-native compiler.

Our benchmarks show that `SUIF/MachSUIF` compiler is competely unapplicable for producing efficient code. This is largerly due to inappropriate instruction selection techniques and lack of optimizations.

Regarding the efficiency of generated code, we saw that generally `gcc` with optimizations on beats all the other retargetable tools. If optimizations are turned off in all tools, `lcc` shows best performance. `VPO` has shown quite irregular performance — on some benchmarks it produces the best code of all, while on others it lose even to non-optimizing `lcc` compiler.

However as a result of auxilliary testing we discovered "contradictionary" benchmarks that are not fit into conclusion given above:

1. `lcc` beats all retargetable tools on Objective Caml [4] garbage collector implementation (30% better than `gcc`) on Intel Pentium III
2. `VPO` beats all retargetable tools on certain implementation of Symbol Ranking text compression algorithm (5 *times* better than `gcc`) on Sun SPARC

Finally we can see that platform-specific Intel compiler outperforms all retargetable tools.

As the ease of retargeting, `lcc` turned out to be the best of all considered tools. `gcc` and `VPO` on the whole show same level of retargetability, although `gcc` is much better documented. `SUIF/MachSUIF` is less retargetable because it is necessary to rewrite codegenerator manually to retarget it.

We conclude that none of the methods considered allows to build a retargetable code generator that can directly be utilized for co-design purposes.

We also see the importance of instruction selection — `lcc`, a non-optimizing compiler with good instruction selection algorithm based on BURS [3,7,13,24, 25] shows quite good performance.

However, good instruction selection is not enough for obtaining optimized code. `VPO` outperforms `lcc` on majority of tests.

This research shows the directions for further development in co–design and code generation area. Easily retargetable, optimizing compilers are vital for hardware-software co-design, but we see that techniques for building them are yet to be created.

Acknowledgments. We would like to thank Mikhail Smirnov and Eugene Vigdorchik — our colleagues in OOPS team of System Programming Department in Saint Petersburg State Unievrsity — for the invaluable discussions and assistance in obtaining the results presented in this paper.

[4] http://caml.inria.fr/index-eng.html

References

1. Alfred V.Aho, S.C.Johnson. Optimal Code Generation for Expression Trees. Journal of the ACM, Vol. 23, No. 3, July 1976, pp. 488–501
2. Alfred V.Aho, Ravi Sethi. Compilers: Principles, Techniques and Tools. Addison-Wesley Pub Co., Nov. 1985
3. Alfred V.Aho, Steven W.K.Tjiang. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, Oct. 1989, pp. 491–516
4. Dmitry Boulytchev, Eugene Vigdorchik, Dmitry Lomov, Mikhail Smirnov. Retargetable Tools for Efficient Code Generation, Technical Report, St.Petersburg State University, January 2001, http://oops.tepkom.ru/eval.html
5. Hubert Comon, Max Dauchet et al. Tree Automata Techniques and Applications. http://l3ux02.univ-lille3.fr/˜tommasi/TATAHTML/main.html
6. H. Emmelmann, F.W.Schröer, R.Landwehr. BEG — a Generator for Efficient Back Ends. Proceedings of the SIGPLAN'89 Conference on Programming Languages Design and Implementation, 1989, pp. 227–237
7. M. Anton Ertl. Optimal Code Selection in DAGs. Proceedings of the 26th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 1999, pp. 242–249
8. Jack W. Davidson, Steve G.Losen, Norman Ramsey. VPO Code-Generation Interfaces. Department of Computer Sciences University of Virginia, 1998, http://www.cs.virginia.edu/zephyr/vpoi
9. Christopher W.Fraser, David R.Hanson. A Retargetable C Compiler: Design and Implementation. Addison-Wesley Pub Co., Jan. 1995
10. Christopher W.Fraser, David R.Hanson. A Retargetable Compiler for ANSI C. ACM SIGPLAN Notices, Vol. 26, No. 10, Oct. 1991, pp. 29–43
11. Christopher W.Fraser, David R.Hanson. A Code Generation Interface for ANSI C. Software — Practice and Experience Vol. 21, No. 9, Sept. 1991, pp. 963–988
12. Christopher W.Fraser, David R.Hanson. Simple register spilling in retargetable compiler. Software — Practice and Experience, Vol. 22, No. 1, Jan. 1992, pp. 85–99
13. Christopher W.Fraser, David R.Hanson, Todd A.Proebsting. Engeneering a simple, efficient code generator generator. ACM Letters on Programming Languages and Systems, Vol. 1, No. 3, Sep. 1992, pp. 213–226
14. Mahadevan Ganapathi, Charles N. Fischer. Affix grammar driven code generation. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, Oct. 1985, pp. 560–599
15. Silvina Hanono, Srinivas Devadas. Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator. Proceedings of the 35th ACM/IEEE Annual Conference on Design Automation, 1998, pp. 510–515
16. David R. Hanson. Early Expirience with ASDL in lcc. http://www.cs.princeton.edu/software/lcc/doc
17. Seh-Woong Jeong, Fabio Somenzi. A New Algorithm for the Binate Covering Problem and Its Application to the Minimization of Boolean Relations. Proceedings of the 1992 IEEE/ACM International Conference on Computer–Aided Design, 1992, pp. 417–420
18. Monika Lam et al., An Overview of the SUIF2 Compiler Infrastructure. Computer Systems Laboratory, Stanford University, 2000, http://suif.stanford.edu/suif/suif2

19. Rainer Leupers, Peter Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. Proceedings of the 1997 European Design and Test Conference
20. Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. Proceedings of 1995 IEEE/ACM International Conference on Computer–Aided Design, 1995, pp. 393–399
21. Robert Morgan. Building an optimizaing Compiler. Digital Press, Feb. 1998
22. Steven Muchnik. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, July 1997
23. Carsten Müllr. Code Selection from Directed Acyclic Graphs in the Context of Domain Specific Digital Signal Processors. Technical Report, Humboldt-Universität zu Berlin, August 10, 1994
24. Eduardo Pelegrí-Llopart, Susan L.Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. Proceedings of the conference on Principles of programming languages, 1988, 294–308
25. Todd A.Proebsting. BURS Automata Generation. ACM Transactions on Programming Languages and Systems, Vol. 17, No. 3, May 1995, pp. 461–486
26. Norman Ramsey, Mary F. Fernandez. Specifying Representation of Machine Instructions. ACM Transactions on Programming Languages and Systems. Vol. 19, No. 3, May 1997, pp. 492–524
27. Norman Ramsey, Jack W. Davidson. Specifying Instructions' Semantics Using λ-RTL (Interim Report). University of Virginia, July 11, 1999, `http://www.cs.virginia/edu/zephyr/csdl/lrtlindex.html`
28. Michael D. Smith, Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries and Optimizations. Division of Engineering and Applied Sciences, Harvard University, 2000, `http://www.eecs.harvard.edu/hube/research/machsuif.html`
29. Michael D. Smith, Glenn Holloway. A User's Guide to the Optimization Programming Interfaces. Division of Engeneering and Applied Sciences, Harvard University, 2000, `http://www.eecs.harvard.edu/hube/research/machsuif.html`
30. Using and Porting GNU Compiler Collection (GCC), `http://gcc.gnu.org/onlinedocs/gcc_toc.html`
31. Bert-Steffen Visser. A Framework for Retargetable Code Generation Using Simulated Annealing. Proceedings of the 25th Euromicro Conference, 1999

Conceptual Data Modeling: An Algebraic Viewpoint

Kazem Lellahi

LIPN, UPRES-A 7030 C.N.R.S
Université Paris 13, Institut Galilée,
93430 Villetaneuse France
`kl@lipn.univ-paris13.fr`, fax +33 (0)1 4826 0712

Abstract. Conceptual modeling of a system consists of giving a structured form of information in a way it captures, as much as possible, the semantics of real word objects. The most popular conceptual model for designing operational databases is the *Entity-Relationship* (ER) model. This model has evolved into models for designing object-oriented system in general. However, despite some formalization attempts, most conceptual techniques remain rather informal. Our aim in this paper is to provide a formal algebraic methodology for conceptual modeling. In the paper we apply our methodology to ER-model but we claim that it is applicable for object modeling with a slight modification. We believe that our approach can help designers in schema validation.

Keywords: algebraic semantics, algebraic specification, conceptual model, meta-modeling.

1 Introduction

The most widely-used conceptual model for designing an operational database is the Entity Relationship model, or ER-model, originally introduced by Chen [4]. The main generic concepts of this model are *entity type*, *relationship type*, *attribute* and *constraint*. An entity type represents a set of real-world objects and a relationship type relates two or several entity types. A property of an entity (or a relationship) type is called an *attribute*. Each attribute, entity type, or relationship type is recognized by a unique identifier called its *name*. Roughly speaking the conceptual schema of an application is the collection of these inter related names. In order to capture more semantics of real-world objects and their relationships, those names are enriched with some constraint declarations. A suitable choice of those names and constraints give us information about the nature (or the semantics) of the application. In general setting, constraints are logical formula about entities, relationships and attributes. The most widely-used constraints are *domain constraint*, *key constraint* and *cardinality constraint*. A domain constraint concerns an attribute, it specifies the *type* of that attribute. An attribute can have a simple type, a record type, or a collection (i.e. a set, a bag or a list) type. A key constraint concerns one entity type and its attributes.

A cardinality constraint concerns an entity type and a relationship type in which that entity type participates. In ER-literature, other kind of constraints have been proposed which concern several relationships types and several entity types. Attributes, entity types, relationship types and constraints provide a specification of the application, representing its *conceptual level*. Several approaches have been proposed for representing this specification graphically. Such a graphical representation is called an ER-*diagram*. There is often a confusion between the schema itself and its ER-diagram. One of the most important steps of designing an operational database is to determine its ER-*diagram*. A CASE (Computer Aided Software Design) tool helps to draw an ER-diagram and generate automatically an operational database schema[1]. Real world objects of an application at a moment of time may be viewed as the *physical level* of the application. It forms a *valid instance* of the ER-diagram. In the majority of proposed approaches, the border between conceptual and physical levels is blurred. This paper proposes an algebraic framework tracing this border neatly. To do this, we define formally, by operations and axioms, the conceptual level called ER-*schema*. Axioms are algebraic counter parts of constraints, and a valid instance is a finite model of these operations and axioms. To define formally the physical level, we interpret each attribute by a set of values called its *domain* and then we naturally extend this interpretation to entity types and relationship types. Note that the domain of an attribute in this approach can be a set of any kind of values, i.e., atomic values, collection values or tuples (records). This separation of conceptual and physical level aligns our approach with classical algebraic specification of data types. Following traditional algebraic approaches, the class of all valid instances of a schema is called its "loose semantics". Moreover, we prove that the design process itself can be structured as a schema — the *meta-schema* — such that the loose semantics of the meta-schema is the class of all well-formed schemas. In a practical point of view the implementation of the meta-schema can help designers in schema validation.

The remainder of the paper is organized as follows. Section 2 briefly describes the basic mathematical background and develops `(max,min)`-cardinality in a general and formal way. Section 3 introduces our formal definition of an ER-schema and compares it with classical approaches. We show how a schema can be seen as an algebra of a specification. Section 4 introduces an instance of such a schema in the style of denotational semantics. Section 5 explores a particular schema called the meta-schema, and proves that each valid instance of the meta-schema is an ER-schema in our sense. Section 6 briefly compares our results with some related works and draws a brief conclusion and indications to further research. Proofs of facts and theorem have been simply sketched.

2 Constraints

For a partial function $f : X \to Y$ we call X its *source* and Y its *target*, and we denote by $def(f)$ its definition domain. The function f is called:

[1] For example, POWER PC (a Sybase product) transforms an ER-diagram into a relational schema coded in SQL.

- *finite* iff the definition domain of f is a finite set,
- *total* iff $def(f) = X$, and *surjective* iff $f(def(f)) = Y$,

For two finite functions with the same source, $f : X \to Y$ and $g : X \to Z$, we say:

- f and g are *exclusive*, denoted $f \cap g = \emptyset$, iff $def(f) \cap def(g) = \emptyset$,
- f and g *cover* $X_1 \subseteq X$, denoted $f \cup g = X_1$, iff $def(f) \cup def(g) = X_1$,
- f is *less defined than* g, denoted $f \sqsubseteq g$, iff $f(x) = g(x)$ for all $x \in def(f)$.

It is self evident that the problem of checking each of above constraints for finite partial functions is a decidable problem.

Let *nat* be the set of nonnegative integers and $\omega = nat \cup \{N\}$, where N is a symbol not in *nat*. We extend the usual order of *nat* to ω by $n < N$ for all $n \in nat$. Ordered pairs $(0, k)$ and (k, N) are said to be *basic cardinalities* if $k \in nat$. For instance, the classical `(min, max)`-cardinalities $(0, 1)$, $(0, N)$ and $(1, N)$ are basic. Now, we specify two binary operations $\wedge$ and $\vee$ and we define the set `CARD` of all cardinalities as follows:

- every basic cardinality is a cardinality, and
- if c_1 and c_2 are cardinalities, then so are $c_1 \wedge c_2$ and $c_1 \vee c_2$.

The semantics of each cardinality c, denoted $[\![c]\!]$, is defined as follows:

$$\begin{aligned}
&[\![(0,0)]\!] = \emptyset;\\
&[\![(0,k)]\!] = \{x \mid x \le k\} \text{ , where } k > 0;\\
&[\![(k,N)]\!] = \{x \mid k \le x \text{ and } x \ne N\}, \text{ where } k > 0;\\
&[\![c_1 \wedge c_2]\!] = [\![c_1]\!] \cap [\![c_2]\!];\\
&[\![c_1 \vee c_2]\!] = [\![c_1]\!] \cup [\![c_2]\!].
\end{aligned}$$

As a result, *relaxed* cardinality and *int-cardinality* constraints (following [12], [8]), are cardinalities in our sense. In particular, the classical cardinality $(1, 1)$ is the cardinality $(0, 1) \wedge (1, N)$, and for any a and b in *nat* such that $0 \le a \le b < N$, the interval cardinality (a, b) is the expression $(a, N) \wedge (0, b)$. But our cardinality expressions also define other kinds of cardinalities. For example, the expression $(0, k) \vee (k_1, N)$ with $k < k_1$ is a new kind of cardinality that we call "at most k or at least k_1". It corresponds semantically to the set $\{n \mid 0 \le n \le k\} \cup \{n \mid k_1 \le n\}$. Roughly speaking, `CARD` can be viewed as a data type specified by two operations $\wedge$ and $\vee$ and freely generated by basic cardinality. Then $[\![\,]\!]$ defines a kind of algebra of that specification.

3 Conceptual Schema

Conceptual modeling of data obeys some rules called *meta-rules*. Meta-rules concern generic concepts and are independent of any applications. For instance, generic concepts of entity-relationship model concern entity, attribute, relationship, cardinality and label (role). Meta-rules for this model may include the following:

Rule 1 : Each entity type participating in a relationship type is accompanied with a `(min,max)`-cardinality and a role (optional).
Rule 2 : Each attribute has a description which determines its type.
Rule 3 : Some attributes of an entity type may be declared as key attributes.
Rule 4 : Every relationship type is at least binary, and If an entity type participates in a relationship type twice, the corresponding roles must be different.
Rule 5 : Every entity type has at least one attribute.
Rule 6 : Every attribute name is either an entity type attribute name or a relationship type attribute name.
Rule 7 : The same attribute name can not be used in an entity type and in a relationship type both (optional rule).
Rule 8 : The same attribute name can not be used in two different entity types (optional rule).

The core of a conceptual data modeling tool consists of an implementation of such rules. Any formalization of such rules can help designers of such systems. In what follows, we shall give a rigorous formalization of these rules in terms of simple mathematical concepts given in the previous section.

Let ATT, ENT, REL, and LAB be enumerable pairwise disjoint sets which have no common element with CARD. Elements of these sets are identifiers representing names of concepts. We call an element of ATT, ENT, REL, and LAB an *attribute name*, an *entity type name*, *a relationship type name* and a *label* (or *role*), respectively. We also consider the set $\mathtt{DESC} = \{\mathtt{mono,\ multi,\ composite}\}$ which is supposed to be disjoint from all other above sets.

Definition 1 (Entity-Relationship schema) A *conceptual* ER-*schema* (or a ER-schema) is a tuple $\mathcal{S} = (d\_att, desc\_att, e\_att, k\_att, r\_ent, r\_att)$ such that

$d\_att : \mathtt{ATT} \to \mathtt{DESC}$,
$desc\_att : \mathtt{ATT} \to \overline{\mathtt{ATT}}$, where $\overline{\mathtt{ATT}} = \mathtt{ATT} \cup set\ \mathtt{ATT}$
$e\_att : \mathtt{ATT} \to \mathtt{ENT}$,
$k\_att : \mathtt{ATT} \to \mathtt{ENT}$,
$r\_att : \mathtt{ATT} \to \mathtt{REL}$, and
$r\_ent : \mathtt{REL} \times \mathtt{ENT} \times \mathtt{LAB} \to \mathtt{CARD}$

are partial functions satisfying the following conditions:

2 $d\_att(A) \in \{\mathtt{mono}, \mathtt{multi}\} \Rightarrow \mathtt{desc\_att(A)} = \mathtt{A}$,
$d\_att(A) = \mathtt{composite} \Rightarrow desc\_att(A) \in \overline{\mathtt{ATT}}$ and $A \notin desc\_att(A)$.
3 $k\_att \sqsubseteq e\_att$.
4 $(R, E_1, l_1) \in def(\mathtt{r\_ent}) \Rightarrow$
$(\exists E_2, \exists l_2\ (R, E_2, l_2) \in def(\mathtt{r\_ent}) \wedge ((E_1 \neq E_2) \vee (l_1 \neq l_2))$.
5 $def(\mathtt{e\_att}) \neq \emptyset$.
6 $\mathtt{e\_att} \cup \mathtt{r\_att} = def(\mathtt{d\_att})$.
7 $\mathtt{e\_att} \cap \mathtt{r\_att} = \emptyset$. ■

To see the connection between this definition, the above rules and the classical definition of the ER-diagram of an application, we show that a given application can be specified by giving a schema. Let $\mathcal{A} = def(\mathtt{d\_att})$, $\mathcal{E} = im(\mathtt{e\_att})$,

$\mathcal{R} = proj\_1(def(\texttt{r\_ent}))$, $\mathcal{L} = proj\_3(def(\texttt{r\_ent}))$, and $\mathcal{C} = im(\texttt{r\_att})$ be the sets of attribute names, entity types names, relationship types name, roles and `(min, max)`-cardinalities of the application, respectively. Then, the definition provides a model of the rules as follows:

- $d\_att(A)$ = `mono`, `multi` or `composite` means that A is a mono-valued, multi-valued, or a composite attribute;
- `desc_att` describes a simple attribute by itself and a composite attribute by a set of attributes;
- $e\_att(A) = E$ means that A is an attribute of entity type E, and $k\_att(A) = E$ means that A is a key attribute of E;
- $r\_att(A) = R$ means that A is an attribute of relationship type R;
- $r\_ent(R, E, l) = c$ means that entity type E participates in relationship type R with cardinality c and role r.

Then, conditions 2-7 correspond to rules 2-7, and Rule 1 and 8 correspond to functionality of `r_ent` and `e_ent`, respectively.

The above description can be expressed alternatively as follows. Let us define the following algebraic specification:

Spec `ER_Sch`
Sorts: `ENTITY, RELSHIP, ATTR,` $\overline{\texttt{ATTR}}$`, LABEL, CARDINALITY, DESCATT`
Ops:
```
D_Att : ATTR → DESCATT (partial),
Desc_Att : ATTR → ATTR‾ (partial),
E_Att : ATTR → ENTITY (partial),
K_Att : ATTR → ENTITY (partial),
R_Att : ATTR → RELSHIP (partial)
R_Ent : RELSHIP ENTITY LABEL → CARDINALITY (partial)
```
Axs:
```
E,E':ENT,R:RELSHIP,A:ATTR,l,l': LABEL,c,c':CARDINALITY,D:DESCATT
 ∀A  K_Att(A) = E ⇒ E_Att(A) = E,
 ∀R ∃ E,  ∃ E',  ∃ c,  ∃ c',  ∃l,  ∃l'
   (R_Ent(R, E, l) = c) ∧ (R_Ent(R, E', l') = c')∧((E≠E')∨(l≠l')),
 ∀E ∃A  E_Att(A) = E,
 ¬( ∃A ∃E ∃E' ((E_Att(A) = E)∧(R_Att(A)=E')))
  ∀A ∃E ((E_Att(A) = E) ∨(R_Att(A) = E')).
```

Fact 1 (schema as algebra) Any ER-schema can be viewed as a "finite partial model" $[\![\,]\!]$ of the specification `ER_Sch` such that:

$[\![\texttt{ATTR}]\!] = \texttt{ATT}$,	$[\![\texttt{RELSHIP}]\!] = \texttt{REL}$
$[\![\texttt{ENTITY}]\!] = \texttt{ENT}$	$[\![\texttt{DESCATT}]\!] = \texttt{DESC}$
$[\![\overline{\texttt{ATTR}}]\!] = \texttt{ATT} \cup \texttt{set ATT}$	$[\![\texttt{LABEL}]\!] = \texttt{LAB}$
$[\![\texttt{CARDINALITY}]\!] = \texttt{CARD}$	

$[\![\texttt{D\_Att}]\!](A) \in \{\texttt{mono}, \texttt{multi}\} \Rightarrow \texttt{desc\_Att}(A) = A$
$[\![\texttt{D\_Att}]\!](A) = \texttt{composite} \Rightarrow \texttt{desc\_Att}(A) \in \overline{\texttt{ATT}}$ and $A \notin \overline{\texttt{ATT}}$.

Conversely any finite partial model of `ER_Sch` satisfying the two last above conditions defines an ER-schema (in the above notation, for a sort s, $[\![s]\!]$ stands for classical notation $[\![\,]\!]_s$, and for an operation f, $[\![f]\!]$ stands for $f^{[\![\,]\!]}$). ■

Note: Some authors [6] allow attribute names to be overloaded in an ER-diagram. This excludes optional rules 7 and 8. To do so in our formalism, we have to consider e_att, k_att and r_att as multi-valued functions (binary relations) and change Definition 1 and Fact 1 mutatis-mutandis. To this end, for any multi-valued function $f : X \to \mathcal{P}_f(Y)$ and any $P \subseteq X$, we define $def(f) = \{x \in X \mid f(x) \neq \emptyset\}$ and $f(P) = \bigcup_{x \in P} f(x)$. We delete from Definition 1 the constraint $e\_att \cap r\_att = \emptyset$. In this way our formalism stands for those multi-valued functions without any change.

In the sequel we denote by $\mathcal{M}odel$(ER_Sch) the collection of models of ER_Sch determined by converse part of the above fact.

4 Instances and Loose Semantics

A *base interpretation* of an ER-schema $\mathcal{S}$ consists of associating a set $[\![A]\!]$ with each attribute name $A \in \mathcal{A}$. The set $[\![A]\!]$ is called the *domain* of A and often written as $dom(A)$. Usually $dom(A)$ is the set of concrete *values* of a type. This type is often a basic type like `int`, `string`, `boolean`. But in most general case, it can also be a record type or a collection type (`sets`, `bags`, `lists`). More precisely, we consider the following type system:

```
BASE :: = int | string | bool | ...
coll :: = set | bag | list
T :: = BASE |coll BASE
Type ::= T | {T, ..., T}.
```

Now, we interpret each attribute A by a type denoted $[\![A]\!]$ such that if `d_att = mono` then $[\![A]\!] \in$ `BASE`, if `d_att = multi` then $[\![A]\!] \in$ `coll BASE`, and if `d_att = composite` then $[\![A]\!] \in$ `{T, ...,T}`.

Every base interpretation of $\mathcal{S}$ can be extended in a "canonical" way in order to interpret entity and relationship types of $\mathcal{S}$ too. The extension is defined as follows:

- Each entity type E with attributes $\{A_1, \cdots, A_n\}$ is interpreted by the set $[\![E]\!]$ of all $\{A_1, \cdots, A_n\}$-tuples over $[\![D_i]\!] = dom(A_i)$ $(1 \leq i \leq n)$. We view such a tuple as a function $e : \{A_1, \cdots, A_n\} \to \bigcup_{1 \leq i \leq n} [\![D_i]\!]$ such that $e(A_i)$, denoted $e.A_i$, is an element of $[\![D_i]\!]$. Each element e of $[\![E]\!]$ is called an *entity* of E and each $e.A_i$ is an *attribute value* participating in e.
- Each relationship type R with attributes $\{A_1, \cdots, A_k\}$ and entity types $\{E_1, \cdots, E_n\}$ is interpreted by the set $[\![R]\!]$ of all $\{A_1, \cdots, A_k, E_1, \cdots, E_n\}$-tuples over $[\![D_i]\!] = dom(A_i)$ $(0 \leq i \leq k)$ and $[\![E_j]\!]$ $(1 \leq j \leq m)$. Thus $[\![R]\!]$ is the set of functions $r : \{A_1, \cdots, A_k, E_1, \cdots, E_m\} \to (\bigcup_{1 \leq i \leq k} [\![D_i]\!]) \cup (\bigcup_{1 \leq j \leq m} [\![E_j]\!])$ such that $r(A_i)$ $(1 \leq i \leq k)$ is in $[\![D_i]\!]$ and $r(E_j)$ $(1 \leq j \leq m)$ is in $[\![E_j]\!]$. Each element r of $[\![R]\!]$ is called a *relationship* of R. Each $r(A_i)$, denoted $r.A_i$, is an *attribute value* participating in r and each $r(E_j)$, denoted $r.E_j$, is an entity participating in r.

Note: The above interpretations of $[\![E]\!]$ and $[\![R]\!]$ take into account the fact that sets of attributes and entity types are unordered sets at the conceptual level. Traditionally those sets are implicitly regarded as ordered sets. In that case one can define $[\![E]\!]$ as $[\![D_1]\!]\times\cdots\times[\![D_n]\!]$, and $[\![R]\!]$ as $[\![D_1]\!]\times\cdots\times[\![D_k]\!]\times[\![E_1]\!]\times\cdots\times[\![E_m]\!]$. Then $e.A_i$ corresponds to usual i-th component of the tuple e.

Definition 2 (Instance) An *instance* $\mathcal{I}$ of schema $\mathcal{S}$ consists of:

- a base interpretation $[\![\ ]\!]$,
- for each $E \in \mathcal{E}$, a finite subset $[\![E]\!]_{\mathcal{I}}$ of $[\![E]\!]$, and
- for each $R \in \mathcal{R}$, a finite subset $[\![R]\!]_{\mathcal{I}}$ of $[\![R]\!]$. ■

Definition 3 (Constraint satisfaction and loose semantics) Let $\mathcal{I}$ be an instance of an ER-schema and *r-ent(R, E,l) = c*. We say that $\mathcal{I}$ *satisfies* the cardinality constraint c for R and E, denoted $R \models_{\mathcal{I}} (E, c)$, iff $|e_R| \in [\![c]\!]$, where $|e_R|$ is the number of elements $r \in [\![R]\!]_{\mathcal{I}}$ in which e participates.
We say that an instance $\mathcal{I}$ for ER-schema satisfies the key constraints for entity type E iff $K = k\_att(E)$ satisfies:

$$\forall e \in [\![E]\!]_{\mathcal{I}}, \forall e' \in [\![E]\!]_{\mathcal{I}}, (\forall A \in K, e.A = e'.A) \Longrightarrow e = e'.$$

A *valid* instance[2] of $\mathcal{S}$ is an instance satisfying all constraints of $\mathcal{S}$. The collection of all valid instances of $\mathcal{S}$ is called the *loose semantics* of $\mathcal{S}$. ■

The loose semantics of a schema $\mathcal{S}$ will be denoted $\mathcal{L}oose(\mathcal{S})$. It is clear that the empty instance is valid. A schema is called *invalid* if its loose semantics contains only the empty instance. Checking whenever an instance is valid is a hard problem in general setting. However, presence of some structural configurations may make this checking easier [5,9,12].

5 The ER-Meta Model

In this section we shall introduce a particular schema META_ER, called *the meta-schema*, in a way that each valid instance of the meta-schema is a schema, and vice-versa. In fact, the meta-schema is obtained by organizing, as a schema, the meta-concepts used in Definition 1.

In a practical point of view implementing the meta schema corresponds to a a part of a design tool. We show that a SQL-like database language can serve as a host language for such an implementation.

Meta schema
We consider the following finite sets:
meta_$\mathcal{A}$ = {D_name, Desc_name, A_name, E_name, R_name, L_name, Card},
meta_$\mathcal{E}$ = { DESCATT, ENTITY, RELSHIP, ATTR, $\overline{\text{ATTR}}$, LABEL},
meta_$\mathcal{R}$ = {D_Att, Desc_Att, E_Att, K_Att, R_Att, R_Ent},
meta_$\mathcal{L}$ = {blank},
meta_$\mathcal{C}$ = {(0, 1), (0, N), (1, N), (2, N)}.
and we suppose that these sets are subsets of ATT, ENT, REL, LAB, and CARD,

[2] Some authors say *satisfiable schema* or *consistent schema*.

respectively [3]. Then, we define the finite functions `meta_e_att`, `meta_k_att`, `meta_r_att` as follows:

```
meta_e_att, meta_k_att, meta_r_att: ATT → ENT
  meta_e_att(A) = meta_k_att(A) =
    {D_name ↦ DESCATT, Desc_name ↦ $\overline{\texttt{ATTR}}$, A_name ↦ ATTR, E_name ↦ ENT,
    R_name ↦ RELSHIP, L_name ↦ LABEL, Card ↦ undefined}
```

$$\texttt{meta\_r\_att}(A) = \begin{cases} \texttt{R\_Ent} & \text{if } A = \texttt{Card}, \\ \texttt{undefined} & \texttt{otherwise} \end{cases}$$

The definition domain of `meta_e_att` and `meta_k_att`, and `meta_r_att` are finite sets `meta_`$\mathcal{A}$ and `meta_`$\mathcal{E}$, respectively. We suppose that all meta attribute are simple and mono-valued. This define two functions:

`meta_d_att : ATT → DESC` and `meta_desc_att : ATT →` $\overline{\texttt{ATT}}$,

with finite definition domains `meta_`$\mathcal{A}$.

Now, we define the function `meta_r_ent: REL × ENT × LAB → CARD` by the following table:

meta_r_ent	REL	ENT	LAB	CARD
	E_Att	ATTR	blank	(0, 1)
	E_Att	ENT	blank	(1, N)
	K_Att	ATTR	blank	(0, 1)
	K_Att	ENT	blank	(0, N)
	R_Att	ATTR	blank	(0, 1)
	R_Att	RELSHIP	blank	(0, N)
	R_Ent	RELSHIP	blank	(2, N)
	R_Ent	LABEL	blank	(1, N)
	R_Ent	ENT	blank	(0, N)
	Desc_att	ATTR	blank	(1, 1)
	Desc_att	$\overline{\texttt{ATTR}}$	blank	(0, N)

Fact 2 `META_ER = (meta_d_att, meta_desc_att, meta_k_att, meta_r_ent, meta_r_att)` is an ER-schema that we call the *meta-schema* of ER-model. ■

Meta instances

In order to define a meta instance we extend our types as follows:

```
IDENT = ATT | ENT | REL | LAB | CARD
$\overline{\texttt{ATT}}$ ::= ATT | set  ATT
Meta_type ::= IDENT | $\overline{\texttt{ATT}}$
Type* ::= Type | Meta_type.
```

Now we define a base interpretation of `META_ER` as follows:

[3] Our definition of schema uses a role for every entity type. In fact, however, roles are only needed when two entity types participate in the same relationship type. We consider `blank` as a non printable element of `LAB` meaning an unnecessary role.

$$\begin{array}{ll}
[\![\texttt{A\_name}]\!] = \texttt{ATT} & [\![\texttt{E\_name}]\!] = \texttt{ENT} \\
[\![\texttt{R\_name}]\!] = \texttt{REL} & [\![\texttt{L\_name}]\!] = \texttt{LAB} \\
[\![\texttt{D\_name}]\!] = \texttt{DESC} & [\![\texttt{Desc\_name}]\!] = \overline{\texttt{ATT}} \\
[\![\texttt{Card}]\!] = \texttt{CARD} &
\end{array}$$

The extension of that base interpretation to meta-entities and meta-relationships is defined by

$$\begin{array}{lllll}
[\![\texttt{DESCATT}]\!] & = \texttt{DESC} = & [\![\texttt{D\_Att}]\!] & & \\
[\![\texttt{ENTITY}]\!] & = \texttt{ENT} = & [\![\texttt{E\_Att}]\!] & = [\![\texttt{K\_Att}]\!] & \\
[\![\texttt{ATTR}]\!] & = \texttt{ATT} = & [\![\texttt{R\_Att}]\!] & & \\
[\![\overline{\texttt{ATTR}}]\!] & = \overline{\texttt{ATT}} = & [\![\texttt{Desc\_Att}]\!] & & \\
[\![\texttt{LABEL}]\!] & = \texttt{LAB} & [\![\texttt{RELSHIP}]\!] & = \texttt{REL} & \\
 & & [\![\texttt{R\_Ent}]\!] & = [\![\texttt{REL}]\!] \times [\![\texttt{ENT}]\!] \times [\![\texttt{LAB}]\!] \times [\![\texttt{CARD}]\!] &
\end{array}$$

Fact 3 Every schema $\mathcal{S}$ defines a valid instance $\mathcal{I}_{\mathcal{S}}$ of META_ER on the above base interpretation. ■

In fact META_ER as defined above is a special finite algebra of the specification ER_Sch.

Corollary 1 There is a valid instance $\mathcal{M}$ of META_ER that defines the schema META_ER itself.

Indeed, META_ER is an ER-schema. Hence, it defines a valid instance $\mathcal{M}$ of META_ER. This instance is defined by $[\![\texttt{ATTR}]\!]_{\mathcal{M}} := \texttt{meta\_}\mathcal{A}$, $[\![\texttt{ENTITY}]\!]_{\mathcal{M}} := \texttt{meta\_}\mathcal{E}$, $[\![\texttt{E\_Att}]\!]_{\mathcal{M}} := \texttt{def(meta\_d\_att)}$, and so on. Clearly $\mathcal{M}$ satisfies all constraints of META_ER. We summarize the above corollary by saying :

The meta schema is a valid instance of itself.

Fact 4 Each valid instance $\mathcal{S}$ of META_ER defines a ER-schema $\mathcal{S}_{\mathcal{I}}$. ■

Following Fact 1, each schema can be viewed as a finite model of ER_Sch. Recall that $\mathcal{M}$od(ER_Sch) is the class of all finite models of ER_Sch satisfying two conditions of fact 1, and $\mathcal{L}$oose(META_ER) is the loose semantics of META_ER (see definition 3).

Theorem 1 $\mathcal{L}$oose(META_ER) is isomorphic to $\mathcal{M}$od(ER_Sch).

Indeed, for any valid instance $\mathcal{I}$ of META_ER define $\Psi(\mathcal{I}) = \mathcal{S}_{\mathcal{I}}$, and for any element $\mathcal{S}$ of $\mathcal{M}$od(ER_Sch) define $\Phi(\mathcal{S}) = \mathcal{I}_{\mathcal{S}}$. Then it can be proved that Φ and Ψ are inverse of each other. In particular, $\Psi(\Phi(\texttt{META\_ER})) = \texttt{META\_ER}$. ■

Implementing the meta schema within SQL

Now, we shall show how we can use the database language SQL as a host language for implementing a part of a conceptual design tool for database applications. Roughly speaking, we transform the meta schema META_ER into a relational database schema RelMETA_ER in such a way that every instance of META_ER corresponds to an instance of RelMETA_ER and vice-versa. We implement then RelMETA_ER within SQL adding PRIMARY KEY, FOREIGN KEY, NOT NULL, and global constraints. An instance $\mathcal{S}$ of META_ER can be defined as a set of INSERT commands for RelMETA_ER. This set of commands forms an instance $\mathcal{I}_{\mathcal{S}}$ of RelMETA_ER. The success of these commands tells us that the instance $\mathcal{I}$ is a a valid instance of META_ER.

We consider a SQL implementation with CHECK OPTION. In such a SQL the relational database schema RelMETA_ER is defined as follows:

```
CREATE TABLE D_Att(A_name  CHAR(10), D_name  CHAR(10)  NOT NULL,
     PRIMARY KEY (A_name));
CREATE TABLE desc_Att(A_name  CHAR(10), Desc_name  CHAR(10)  NOT NULL,
     FOREIGN KEY (A_name)REFERENCES D_name ON DELETE CASCADE);
CREATE TABLE E_Att(A_name  CHAR(10), E_name  CHAR(10) NOT NULL,
    PRIMARY KEY (A_name),
     FOREIGN KEY (A_name) REFERENCES D_name ON DELETE CASCADE);
CREATE TABLE  K_Att(A_name CHAR(10), E_name  CHAR(10) NOT NULL
     PRIMARY KEY (A_name),
     FOREIGN KEY (A_name) REFERENCES E_Att ON DELETE CASCADE);
CREATE TABLE R_Ent(R_name CHAR(10), E_name  CHAR(10),
             L_name  CHAR(10) NOT NULL, Card  CHAR(10) NOT NULL,
     PRIMARY KEY (R_name, E_name, L_name),
     CONSTRAINT unknown_E_name CHECK
                  (E_name IN (SELECT E_name  FROM  E_Att)));
CREATE TABLE R_Att(A_name  CHAR(10), R_name CHAR(10),
     PRIMARY KEY (A_name),
     FOREIGN KEY (A_name)  REFERENCES D_name ON DELETE CASCADE,
     CONSTRAINT unknown_R_name
             CHECK (R_name IN (SELECT R_name FROM R_Ent)));

   CREATE ASSERTION  C1  CHECK
   ( NOT EXISTS (SELECT R_name, COUNT(E_name)  FROM  R_Ent
       GROUP BY R_name HAVING COUNT(E_name) < 2));
CREATE ASSERTION  C2  CHECK
   (NOT EXISTS (SELECT A_name  FROM  D_name
       WHERE A_name  NOT  IN
       (( SELECT A_name   FROM  E_Att)
         UNION ( SELECT A_name   FROM  R_Att))));
CREATE ASSERTION  C3  CHECK
    ( NOT EXISTS (SELECT A_name  FROM  E_Att
      WHERE A_name IN ( SELECT A_name  FROM  R_Att))
        AND NOT EXISTS (SELECT A_name  FROM  R_Att
      WHERE A_name IN (SELECT A_name  FROM  E_Att)));
```

Theorem 2 *Each valid* ER*-schema* $\mathcal{S}$ *defines a valid relational database instance* $\psi(\mathcal{S})$ *over the relational schema* RelMETA_ER*. Conversely, each valid instance* I *of* RelMETA_ER *defines a valid* ER*-schema* $\phi(I)$*. Moreover,* ψ *and* ϕ *are inverse of each other; that is* $\psi(\phi(I)) = I$ *and* $\phi(\psi(\mathcal{S})) = \mathcal{S}$.[4]

Proof: In what follows the set of answers of a query q on a database I over the schema RelMETA_ER is denoted $[\![q]\!]_I$.
Let $\mathcal{S}$ be a schema. According to the Fact 3, $\mathcal{S}$ can be seen as an instance $\mathcal{I}_\mathcal{S}$ of META_ER. To define $\phi(\mathcal{S})$ we insert within a unique transaction all elements of $[\![\texttt{D\_Att}]\!]_{\mathcal{I}_\mathcal{S}}$, $[\![\texttt{Desc\_Att}]\!]_{\mathcal{I}_\mathcal{S}}$, $[\![\texttt{E\_Att}]\!]_{\mathcal{I}_\mathcal{S}}$, $[\![\texttt{K\_Att}]\!]_{\mathcal{I}_\mathcal{S}}$, $[\![\texttt{R\_Ent}]\!]_{\mathcal{I}_\mathcal{S}}$ and $[\![\texttt{R\_Att}]\!]_{\mathcal{I}_\mathcal{S}}$ in the relations D_Att, Desc_Att, E_Att, R_Ent and R_Att of RelMETA_ER , respectively. Conversely, let I be a valid instance of RelMETA_ER (i.e. an instance satisfying all constraints and all general ASSERTIONs). We define the schema $\psi(I) = (d\_att, desc\_att, k\_att, r\_ent, r\_att)$ as follows:

$$\begin{array}{llll} d\_Att = & [\![\texttt{SELECT} * \texttt{FROM D\_Att;}]\!]_I & desc\_att = & [\![\texttt{SELECT} * \texttt{FROM Desc\_Att;}]\!]_I \\ e\_att = & [\![\texttt{SELECT} * \texttt{FROM E\_Att;}]\!]_I & k\_att = & [\![\texttt{SELECT} * \texttt{FROM K\_Att;}]\!]_I \\ r\_ent = & [\![\texttt{SELECT} * \texttt{FROM R\_Ent;}]\!]_I & r\_att = & [\![\texttt{SELECT} * \texttt{FROM R\_Att;}]\!]_I \end{array}$$

It is clear that ϕ and ψ are inverse of each other. Moreover, conditions of the definition 1 are respectively equivalent to the success on insert commands.

Therefore if $\mathcal{S}$ satisfies conditions of the definition 1 then $\phi(\mathcal{S})$ is a valid instance of RelMETA_ER. Conversely, if $\mathcal{I}$ is a valid instance of META_ER then $\psi(\mathcal{I})$ satisfies conditions of the definition 1. This completes the proof. ■

Nota: Checking assertions C1 to C3 is equivalent, respectively, to checking that the following queries Q_1 to Q_3 have no answers:

Q_1:
```
SELECT R_name, COUNT(E_name)
  FROM  R_Ent
    GROUP BY R_name
    HAVING COUNT (E_name) < 2;
```

Q_2 :
```
SELECT A_name
  FROM  D_name
   WHERE A_name   NOT IN
    ((SELECT A_name FROM E_Att)
  UNION
     (SELECT A_name  FROM  R_Att));
```

Q_3:
```
(SELECT SELECT A_name   FROM  E_Att)
 INTERSECT (SELECT SELECT A_name  FROM  R_Att);
```

As a result one can replace the assertions by checking emptiness of these queries. This is a good solution when the the current SQL implementation does not support global assertion constraints. For more details on this implementation see [3]

6 Related Works, Conclusion, and Further Research

Formalization and unification of conceptual modeling approaches have been of interest for many authors. Several works extend the original Chen proposal [4]

[4] By a valid instance of a relational database we mean an instance that satisfies all its constraints.

to new concepts, including inheritance and objects [7,6]. For instance, in [11] a formal higher order ER-model has been proposed. In this model the notion of relationship has been extended by introducing a relationship of higher order, permitting to nest relationships. To capture more semantics, a wide variety of constraints have been introduced in [9,2,12]. In [8,12] a formal approach has been proposed for unification of these constraints. An amount of papers are devoted to the delicate problem of checking validity of an ER-schema in presence of constraints [5,9,8]. However, in all these works constraints are specified semantically. Meta modeling is another subject which has been used in various approaches of conceptual modeling, including reverse engineering, schema integration and model transformation. For instance, in [1] a meta model is used for reverse engineering, more precisely for discovering inheritance hidden links in a relational schema. From our point of view, the border between conceptual and physical levels is blurred in the above proposals, and their meta models are ad hoc and lack a formal basis. In this paper we have proposed a formal approach for ER-models. The approach does a neat separation between the specification of conceptual and physical data of an application. Another particularity of the present work is an attempt to distinguish between specification of constraints and their satisfaction. We have proved that our formalism is self-contained in the sense that all schemas are instances of a special schema: the meta schema. Proofs of main results are omitted for space limitation.
Many interesting aspects of conceptual modeling are not developed in this extended abstract, dynamic aspects and data warehousing [10] are among them. It is interesting to give a formal specification of the underlying dynamic system. The technique presented in this paper seems to be applicable for more sophisticated modeling approaches, especially for object modeling and a semantics of UML. We are currently investigating these research directions.

References

1. J. Akoka, I. Comyn-Wattiau, and N. Lammari. Relational Database Reverse Engineering, Elicitation of Generalization Hierarchies. *Advance in Conceptual Modeling, ER'99 Workshops on Evolution and Change in Data Management, LNCS 1727*, pages 173–185, November 1999.
2. J. B. Behm and T.J. Teorey. Relative Constraint in ER Data Model. *ER'93:Entitiy-Relationship Approach, 12th International Conference on the Entity-Relationship approach, LNCS 823*, pages 46–59, December 1993.
3. F. Boufares. Un outil intelligent pour l'analyse des schémas EA. *Research Report 2001-05, Université Paris 13, LIPN.*
4. P.P. Chen. The Entity-Relationship Model — Towards a Unified view of Data. *ACM Transaction On Database System*, 1(1):9–36, March 1976.
5. J. Dullea and Il-Yeol Song. A Taxonomy of Recursive Relationships and Their Structural Validity in ER Modeling. *ER'99 : Conceptual Modeling, 18th International Conference on Conceptual Modeling, LNCS 1728*, pages 384–398, 1999.
6. R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, *The Bejamin Cummings Publishing Company, Inc., Second Edition*, 1994.

7. G. Engels, M. Gogolla, U. Hohenstein, K. Hulsmann, P. Lohr-Richter, G. Saake and H. D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data and Knowledge Engineering*, 9:157–204, 1993.
8. S. Hartmann. On the Consistency Of Int-cardinality Constraints. *ER'98: Conceptual Modeling, 17th International Conference on Conceptual Modeling, LNCS 1507*, Pages 150–163, November 1998.
9. M. Lenzerini and P. Nobili. On the satisfiability of dependency constraints in Entity-Relationship schemata. *Information Systems*, 15(4):453–461, 1990.
10. D. Moody and Kortink. From Entities to Stars, Snowflakes, Constellations and Galixies: A Methodology for Data Warehouse Design. *18th International Conference on Conceptual Modelling Proceedings*, pages 114–130, March 1999.
11. B. Thalheim. The Higher-Order Entity-Relationship Model and DB2. *MFDBS'89: 2nd Symposium on Mathematical Fundamentals of Database Systems, LNCS 364*, pages 382–397, June 26–30 1989.
12. B. Thalheim. Fundamentals of Cardinality Constraints. *ER'92: Entity-Relationship Approach, 11th International Conference on the Entity-Relationship Approach, LNCS 645*, pages 7–23, October 7–9 1992.

Integrating and Managing Conflicting Data

Sergio Greco, Luigi Pontieri, and Ester Zumpano

DEIS, Università della Calabria
87030 Rende, Italy
{greco,pontieri,zumpano}@si.deis.unical.it

Abstract. Data integration aims to provide a uniform integrated access to multiple heterogeneous information sources, designed independently and having strictly related contents. The heterogeneity among sources may range from the hardware and software platforms to the data model and the schema used to represent information.
In this paper we consider data inconsistencies and deal with the integration of possibly conflicting instances coming from different sources and related to the same concept. In order to support the data integration process, we introduce some integration operators for combining data coming by different sources, preserving the information provided by each source separately. Generally, a view obtained by integrating heterogeneous sources could contain instances which are inconsistent, w.r.t. some integrity constraints defined on the integrated view schema. Therefore, we present a technique which allows us to compute consistent answers to queries involving possibly inconsistent data.

1 Introduction

A central topic in database science is the construction of integration systems, designed for retrieving and querying uniformly data stored in multiple information sources. The main problem which integration systems have to deal with is the heterogeneity of their sources, often designed independently for autonomous applications. The problem of integrating heterogeneous sources has been deeply investigated in the fields of multidatabase systems [4], federated databases [22] and, more recently, mediated systems [20,22]. Mediator-based architectures are characterized by the presence of two types of components: *wrappers*, which translate the local languages, models and concepts of the data sources into the global ones, and *mediators*, which take in input information from one or more components below them and provide an integrated view of them [7,16]. Views, managed by mediators, may be virtual or materialized. When a mediator receives a query, it dispatches subqueries to the components below it (wrappers and/or mediators), collects the results and merges them in order to construct the global answer. Mediators have to cope with schema and value inconsistencies that may be present in the information coming from the different sources. The first kind of inconsistency arises when different sources use different schemas to model the same concept, while the second one arises when different sources record different values for the same object [14,21].

In this paper, we focus our attention on the integration of conflicting instances [1,2,6] related to the same concept and possibly coming from different sources. We introduce an operator, called *Merge operator*, which allows us to combine data coming from different sources, preserving the information contained in each of them and a variant of merge operator, i.e. the *Prioritized Merge operator*, which can be employed to combine data using preference criteria Generally, at any level of the architecture, the integrated information may be inconsistent, i.e. it doesn't satisfy some integrity constraints associated with the schema of the mediator. Thus we present a technique, based on the identification of tuples satisfying integrity constraints and on the selection of tuples satisfying the query, which permits us to compute consistent answers, i.e. maximal sets of atoms which do not violate the constraints.

1.1 Background

A *(disjunctive Datalog) rule* r is a clause of the form

$$A_1 \vee \cdots \vee A_k \leftarrow B_1, \cdots, B_m, \mathit{not}\ B_{m+1}, \cdots, \mathit{not}\ B_n, \qquad k+m+n>0,$$

where $A_1, \cdots, A_k, B_1, \cdots, B_n$ are atoms of the form $p(t_1, ..., t_h)$, p is a *predicate* of arity h and the terms $t_1, ..., t_h$ are constants or variables. In the following, we also assume the existence of null value ($\perp$) in the domain of the constants in order to model a lack of knowledge. The disjunction $A_1 \vee \cdots \vee A_k$ is the *head* of r, while the conjunction $B_1, \cdots, B_m, \mathit{not}\ B_{m+1}, \cdots, \mathit{not}\ B_n$ is the *body* of r. We also assume the existence of the binary built-in predicate symbols (comparison operators) which can only be used in the body of rules.

An interpretation M for $\mathcal{P}$ is any subset of the Herbrand base of $\mathcal{B}_{\mathcal{P}}$; M is a model of $\mathcal{P}$ if it satisfies all rules in *ground*($\mathcal{P}$). The (model-theoretic) semantics for positive $\mathcal{P}$ assigns to $\mathcal{P}$ the set of its *minimal models* MM($\mathcal{P}$), where a model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$ [18]. The more general *disjunctive stable model semantics* also applies to programs with (unstratified) negation [9]. Disjunctive stable model semantics generalizes stable model semantics, previously defined for normal programs [8]. For any interpretation I, denote with $\mathcal{P}^I$ the ground positive program derived from *ground*($\mathcal{P}$) by 1) removing all rules that contain a negative literal *not a* in the body and $a \in I$, and 2) removing all negative literals from the remaining rules. An interpretation M is a (disjunctive) stable model of $\mathcal{P}$ if and only if $M \in \mathrm{MM}(\mathcal{P}^M)$. For general $\mathcal{P}$, the stable model semantics assigns to $\mathcal{P}$ the set SM($\mathcal{P}$) of its *stable models*. It is well known that stable models are minimal models (i.e. $\mathrm{SM}(\mathcal{P}) \subseteq \mathrm{MM}(\mathcal{P})$) and that for negation free programs minimal and stable model semantics coincide (i.e. $\mathrm{SM}(\mathcal{P}) = \mathrm{MM}(\mathcal{P})$). Observe that stable models are minimal models which are 'supported', i.e. their atoms can be derived from the program.

1.2 Extended Disjunctive Databases

Extended Datalog programs extend standard Datalog programs with a different form of negation, known as *classical* or *strong negation*, which can also appear in the head of rules. Thus, while standard programs provide negative information implicitly, extended programs provide negative information explicitly and we can distinguish queries which fail in the sense that they do not succeed and queries which fail in the stronger sense that negation succeeds [9,10,15]. An extended atom is either an atom, say A or its negation $\neg A$. An extended Datalog program is a set of rules of the form

$$A_0 \vee ... \vee A_k \leftarrow B_1, ..., B_m, not\ B_{m+1}, ..., not\ B_n$$

where $k + n > 0$, $A_0, ..., A_k, B_1, ..., B_n$ are extended atoms. A (2-valued) interpretation I for an extended program $\mathcal{P}$ is a pair $\langle T, F\rangle$ where T and F define a partition of $\mathcal{B}_\mathcal{P} \cup \neg\mathcal{B}_\mathcal{P}$ and $\neg\mathcal{B}_\mathcal{P} = \{\neg A | A \in \mathcal{B}_\mathcal{P}\}$. The truth value of an extended atom $L \in \mathcal{B}_\mathcal{P} \cup \neg\mathcal{B}_\mathcal{P}$ w.r.t. an interpretation I is equal to (i) *true* if $L \in T$ and, (ii) $false$ if $A \in F$. Moreover, we say that an interpretation $I = \langle T, F\rangle$ is *consistent* if there is no atom A such that $A \in T$ and $\neg A \in T$. The semantics of an extended program $\mathcal{P}$ is defined by considering each negated predicate symbol, say $\neg p$, as a new symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n the constraint $\leftarrow p(X_1, ..., X_n), \neg p(X_1, ..., X_n)$. The existence of a (2-valued) model for an extended program is not guaranteed, also in the case of negation (as-failure) free programs. For instance, the program consisting of the two facts a and ¬a does not admit any (2-valued) model. In the following, for the sake of simplicity, we shall also use rules whose bodies may contain disjunctions. Such rules, called generalized disjunctive rules, are used as shorthands for multiple standard disjunctive rules. More specifically, a generalized disjunctive rule of the form

$$A_1 \vee ... \vee A_k \leftarrow (B_{1,1} \vee ... \vee B_{1,m_1}), ..., (B_{n,1} \vee ... \vee B_{n,m_n})$$

denotes the set of standard rules

$$A_1 \vee ... \vee A_k \leftarrow B_{1,i_1}, ..., B_{n,i_n} \forall j, i$$

where $1 \leq j \leq n$ and $1 \leq i_j \leq m_j$. Given a generalized disjunctive program $\mathcal{P}$, $st(\mathcal{P})$ denotes the standard disjunctive programs derived from $\mathcal{P}$ by rewriting body disjunctions.

1.3 Queries

Predicate symbols are partitioned into two distinct sets: *base predicates* (also called EDB predicates) and *derived predicates* (also called IDB predicates). Base predicates correspond to database relations defined over a given domain and they do not appear in the head of any rule whereas derived predicates are defined by means of rules. Given a database D, a predicate symbol r and a program $\mathcal{P}$, $D(r)$ denotes the set of r-tuples in D whereas $\mathcal{P}_D$ denotes the program derived from the union of $\mathcal{P}$ with the tuples in D, i.e. $\mathcal{P}_D = \mathcal{P} \cup \{r(t) \leftarrow\ | \ t \in D(r)\}$. In the following a tuple t of a relation r will be also denoted as a fact $r(t)$.

The semantics of $\mathcal{P}_D$ is given by the set of its stable models by considering either their union (*possible semantics* or *brave reasoning*) or their intersection (*certain semantics* or *cautious reasoning*). A (relational) query over a database defines a function from the database to a relation. It can be expressed by means of alternative equivalent languages such as relational algebra, 'safe' relational calculus or 'safe' non-recursive Datalog [19]. In the following we shall use Datalog. Thus, a query is a pair $(g, \mathcal{P})$ where $\mathcal{P}$ is a safe non-recursive Datalog program and g is a predicate symbol specifying the output (derived) relation. Observe that relational queries define a restricted case of disjunctive queries. The reason for considering relational and disjunctive queries is that relational queries over databases with constraints can be rewritten into extended disjunctive queries over databases without constraints.

2 Data Integration

A mediator provides an integrated view over a set of information sources. Each of these sources may be a source database or a database view (virtual or materialized) furnished by another mediator. At each level of the integration system, the information provided by different sources and related to the same concept is combined. The necessity of completing data stored at a source is due to the fact that some information may not be available at that source because it is not modeled within the schema of the source or simply because some instances contain undefined values for some attributes. The way we integrate different sources preserves the information contained in each of them, since we try to complete the information but we never modify that already available. In fact our integration operators, described in the following paragraphs, are designed for reducing the incompleteness and redundancy of information but they do not solve all data conflicts, as such operation could lead to loss of information.

Let us introduce some basic definitions in order to simplify the description of our approach.

We assume that a mediator has its own schema, that we call *mediator schema*, and a set of integrity constraints whose satisfaction means that data are consistent. The mediator schema represents, in an integrated way, some relevant concepts that may be modeled differently within the schemas of different sources. Integrity constraints are first order formulas which must always be true. Although in this paper we only consider functional dependencies, our approach to manage inconsistent data is more general.

Let us adopt the relational model for referring schemas and instances pertaining to the mediator and the sources it integrates.

Notation: Let R be a relation name, then we denote by:

- *attr(R)* the set of attributes of R;
- *key(R)* the set of attributes in the primary key of R;
- *inst(R)* an instance of R (set of tuples).

Moreover, given a tuple $t \in inst(R)$, $key(t)$ denotes the values of the key attributes of t. In the rest of the paper we will use the name of a relation for denoting its content, unless that generates ambiguity.

We assume that relations associated with the same concept have been homogenized [21] with respect to a common ontology, so that attributes denoting the same concepts have the same name and the same domain. We say that two homogenized relations R and S, associated with the same concept, are *overlapping* if $key(R) = key(S)$.

Definition 1. Let $R_1, ..., R_n$ be a set of overlapping relations. A relation R is a *super-relation* of $R_1, ..., R_n$ if the following conditions hold:

- $attr(R) = \bigcup_{i=1}^{n} attr(R_i)$,
- $key(R) = key(R_i) \quad \forall\ i = 1..n$. $\square$

Moreover, if R is a super-relation of $R_1, ..., R_n$, then we say that R_i is a *sub-relation* of R for $i = 1..n$.

A mediator defines the content of any global relation as an integrated view of the information provided by all its sub-relations. Once the logical conflicts due to the schema heterogeneity have been resolved, conflicts may arise, during the integration process, among instances provided by different sources. In particular, the same real-world object may correspond to many tuples (possibly residing in different overlapping relations), that may have the same value for the key attributes but different values for some non-key attribute. In our approach both source relations and integrated views could have inconsistent instances. In particular, it is possible that a relation contains more than one tuple with the same value for the key attributes. A set of tuples with the same key value is called *c-tuple* (cluster of tuples) [1], so in this context we often consider relations as sets of c-tuples.

The content of an integrated view may be looked at as the result of an *integration operator* applied to the overlapping relations which constitute the information sources for that view. For the sake of simplicity, in the rest of the paper we consider binary operators for integrating source relations.

Definition 2. Let R_1 and R_2 be two overlapping relations, then an *integration operator* θ is a binary operator which produces an integrated relation $R_1 \theta R_2$, such that $R_1 \theta R_2$ is a super-relation of R_1 and R_2 and its tuples are obtained by combining the values contained in R_1 and R_2. $\square$

In order to provide a simple framework for characterizing and evaluating the various integration operators, we introduce a binary relationship for comparing two tuples or two relations.

Definition 3. Let DS be a set of attributes and t_1, t_2 be two tuples over DS, then $t_1 \sqsubseteq t_2$ if for each attribute A in DS, $t_1[A] = t_2[A]$ or $t_1[A] = null$. Moreover, given two relations R and S over DS, $R \sqsubseteq S$ if $\forall t_1 \in inst(R)\ \exists t_2 \in inst(S)$ s.t. $t_1 \sqsubseteq t_2$. $\square$

On the basis of Definition 3, we define in the following some intuitive properties for an integration operator.

Definition 4. Let R_1 and R_2 be two overlapping relations and θ be an integration operator, then θ is:

- $\sqsubseteq$-*Complete* if $R_1 \sqsubseteq R_1\theta R_2$ and $R_2 \sqsubseteq R_1\theta R_2$
- $\sqsubseteq$-*Correct* if $\forall t \in inst(R_1\theta R_2)\ \exists t'$ s.t. ($t' \in R_1$ or $t' \in R_2$) and $t' \sqsubseteq t$ $\square$

Informally, if an integration operator is both $\sqsubseteq$-Complete and $\sqsubseteq$-Correct it preserves the information provided by the sources. In fact it could modify some input tuples by replacing null values with not null ones, but all the associations of not null values which were contained in the source relations will be inserted into the result. An important feature of the integration process is related to the way conflicting tuples provided by overlapping relations are combined. Simple examples of integration operators are the union and the natural join. It can be immediately verified that the former is both $\sqsubseteq$-Correct and $\sqsubseteq$-Complete, whereas the latter is $\sqsubseteq$-Correct but not $\sqsubseteq$-Complete. However, the relation obtained by means of the union operator could be quite redundant, since tuples relative to the same real-world instance are not joined at all. A better example of integration operator is constituted by the outer join operator (natural), which is $\sqsubseteq$-Correct and $\sqsubseteq$-Complete, and produces a relation less redundant than the union. In fact some tuples in the outer join relation could be obtained by merging sets of tuples contained in the union relation, i.e. $R_1 \cup R_2 \sqsubseteq R_1 ⟗ R_2$.

In the following section we define two integration operators, namely the *Merge Operator* and the *Prioritized Merge Operator*. The former tries to reduce the information redundancy inside the integrated view but preserves the information provided by both the source relations. On the contrary, the prioritized operator eliminates some inconsistency, since in case of conflicting values it prefers the information contained in the first input relation.

The Merge Operator

Let us first introduce the binary operator Φ which replaces null values occurring in a relation with values taken from a second one. In more detail, given two relations R and S such that $attr(S) \subseteq attr(R)$, the operator is defined as follows:

$$\Phi(R,S) = \{\ t \in R \mid \nexists t_1 \in S\ s.t.\ key(t) = key(t_1)\ \} \cup$$
$$\left\{ t \mid \exists t_1 \in R, \exists t_2 \in S\ s.t. \forall a \in attr(R)\ \left(t[a] = \begin{cases} t_2[a] & \text{if } (a \in attr(S) \wedge t_1[a] = \bot) \\ t_1[a] & \text{otherwise} \end{cases} \right) \right\}$$

Given two overlapping relations S_1 and S_2, the *merge operator*, denoted by $\boxtimes$, integrates the information provided by S_1 and S_2. Let $S = S_1 \boxtimes S_2$, then the schema of S contains both the attributes in S_1 and S_2, and its instance is obtained by completing the information coming from each input relation with that coming from the other one.

Definition 5. Let S_1 and S_2 be two overlapping relations. The *merge operator* is a binary operator defined as follows:

$$S_1 \boxtimes S_2 = \Phi(S_1 ⟕ S_2, S_2) \cup \Phi(S_1 ⟖ S_2, S_1)$$

$\square$

$S_1 \boxtimes S_2$ computes the full outer join and extends tuples coming from S_1 (resp. S_2) with the values of tuples of S_2 (resp. S_1) having the same key. The extension of a tuple is carried out by the operator Φ which replaces null values appearing in a given tuple of the first relation with values appearing in some correlated tuple of the second relation. Thus, the merge operator applied to two relation S_1 and S_2 'extends' the content of tuples in both S_1 and S_2.

Proposition 1. *The merge operator is an integration operator, w.r.t. Definition 2. In fact, let S_1 and S_2 be two overlapping relations, then*

- $attr(S_1 \boxtimes S_2) = attr(S_1) \bigcup attr(S_2)$
- $key(S_1 \boxtimes S_2) = key(S_1) = key(S_2)$, □

Example 1. Consider the relations $S1$ and $S2$ reported in Fig. 1 in which K is the key of the relations and the functional dependency $Title \rightarrow Author$ holds. The relation T is obtained by merging S_1 and S_2, i.e. $T = S_1 \boxtimes S_2$.

K	$Title$	$Author$
1	$Moon$	$Greg$
2	$Money$	$Jones$
3	Sky	$Jones$

S_1

K	$Title$	$Author$	$Year$
3	$Flowers$	$Smith$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

S_2

K	$Title$	$Author$	$Year$
1	$Moon$	$Greg$	$\perp$
2	$Money$	$Jones$	$\perp$
3	Sky	$Jones$	1965
3	$Flowers$	$Smith$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

T

Fig. 1.

□

Let S_1 and S_2 be two overlapping relations, let $K = key(S_1) = key(S_2)$, $A = \{a_1, ..., a_n\} = attr(S_1) \cap attr(S_2) - K$, $B = \{b_1, ..., b_m\} = attr(S_1) - attr(S_2)$ and $C = \{c_1, ..., c_q\} = attr(S_2) - attr(S_1)$. The merge operator introduced in Definition 5 can easily be expressed by means of the following SQL statement (where, given a relation R and a set of attributes $X = X_1, ..., X_t$, the notation $R.X$ stands for $R.X_1, ..., R.X_t$):

```
SELECT  S1.K, S1.B, COALESCE(S1.a1, S2.a1), .., COALESCE(S1.an, S2.an), S2.C
FROM    S1 LEFT OUTER JOIN S2 ON S1.K = S2.K
UNION
SELECT  S2.K, S1.B, COALESCE(S2.a1, S1.a1), .., COALESCE(S2.an, S1.an), S2.C
FROM    S1 RIGHT OUTER JOIN S2 ON S1.K = S2.K
```

where the standard operator `COALESCE`$(a_1, ..., a_n)$ returns the first not null value in the sequence.

Proposition 2.

- $S_1 \boxtimes S_2 = S_2 \boxtimes S_1$ *(commutativity),*
- *if* $inst(S_1)$ *does not contain null values then*
  $S_1 \boxtimes S_1 = S_1$ *(idempotency),*
- $S_1 \sqsubseteq S_1 \boxtimes S_2$ *and* $S_2 \sqsubseteq S_1 \boxtimes S_2$ *(*$\sqsubseteq$*-Completeness),*
- $\forall\ t \in inst(R_1 \boxtimes R_2)\ \exists t'$ *s.t.*
  $(t' \in R_1$ *or* $t' \in R_2)$ *and* $t' \sqsubseteq t$ *(*$\sqsubseteq$*-Correctness).* □

Obviously, given a set of overlapping relations $S_1, S_2, ..., S_n$, the associated super-relation S can be obtained as $S = S_1 \boxtimes S_2 \boxtimes ... \boxtimes S_n$. In other words S is the integrated view of $S_1, S_2, ..., S_n$.

The problem we are considering is similar to the one treated in [21], which assumes the source relations involved in the integration process have previously been homogenized. In particular, any homogenized source relation is a fragment of the global relation, that is it contains a subset of the attributes of the global relation and has the same key K. The technique proposed in [21] makes use of an operator $\bowtie$, called *Match Join*, to manufacture tuples in global relations using fragments. This operator consists of the outer-join of the $ValSet$ of each attribute, where the $ValSet$ of an attribute A is the union of the projections of each fragment on $\{K, A\}$. Therefore, the application of the Match Join operator produces tuples containing associations of values that may not be present in any fragment.

Example 2. We report below the relation obtained by applying the Match Join operator to the relations S_1 and S_2 of Example 1.

K	$Title$	$Author$	$Year$
1	$Moon$	$Greg$	$\perp$
2	$Money$	$Jones$	$\perp$
3	Sky	$Jones$	1965
3	Sky	$Smith$	1965
3	$Flowers$	$Smith$	1965
3	$Flowers$	$Jones$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

T

□

The Match Join operator is $\sqsubseteq$-Complete, but it is not $\sqsubseteq$-Correct, since it mix values coming from different tuples with the same key in all possible ways. As a consequence, when applying the Match Join operator to the source relations of Example 1 we obtain an integrated view T violating the functional dependency *Title* $\to$ *Author*. On the contrary, the merge operator here introduced only tries to derive unknown values and for each cluster of tuples in the derived relation the functional dependencies are satisfied. Observe also that the Match Join operator may not satisfy the idempotent property, i.e. $S_1 \bowtie S_1 \neq S_1$, even if the source relation does not contain null values.

In Table 1 some integration operators are compared, by taking into account the following relevant properties: $\sqsubseteq$-Completeness, $\sqsubseteq$-Correctness, and key consistency preservation. The last property indicates that whenever the input relations are consistent w.r.t. the key functional dependency, i.e. all c-tuples are singleton, also the integrated relation will be consistent w.r.t. the key functional dependency. We can observe that only the union, the outer join (natural) and the merge operator preserve the source information, i.e. they are both $\sqsubseteq$-Correct and $\sqsubseteq$-Complete. However, the relation produced by the last operator contains a lower number of unknown values. In fact $R_1 \cup R_2 \sqsubseteq R_1 \boxtimes R_2$ and $R_1 \sqsupset\!\bowtie\!\sqsubset R_2 \sqsubseteq R_1 \boxtimes R_2$.

Table 1. Properties of various integration operators

operator	$\sqsubseteq$-*Complete*	$\sqsubseteq$-*Correct*	*Preserves Key-Consistency*
union	*yes*	*yes*	*no*
naturaljoin	*no*	*yes*	*yes*
outerjoin	*yes*	*yes*	*no*
matchjoin	*yes*	*no*	*no*
merge	*yes*	*yes*	*no*

3 Answering Queries Satisfying User Preferences

In this section we introduce a variant of the merge operator, which allows the mediator to answer queries according to the user preferences. Preference criteria are expressed by a set of constraints called *preference constraints* which permit us to define a partial order on the source relations.

A *preference constraint* is a rule of the form $S_i \ll S_j$, where S_i, S_j are two source relations. Preference constraints imply a partial order on the source relations. We shall write $S_1 \ll S_2 \ll ... \ll S_k$ as a shorthand of $\{S_1 \ll S_2, S_2 \ll S_3, ..., S_{k-1} \ll S_k\}$. The presence of such constraints requires the satisfaction of preference criteria during the computation of the answer. A priority statement of the form $S_i \ll S_j$ specifies a preference on the tuples provided by the relation S_i with respect to the ones provided by the relation S_j.

The Prioritized Merge Operator

In order to satisfy preference constraints, we introduce an asymmetric merge operator, called *prioritized merge operator*, which gives preference to data coming from the left relation, when conflicting tuples are detected.

Definition 6. Let S_1 and S_2 be two overlapping relations and $S_2' = S_2 \triangleright\!< (\pi_{key(S_2)} S_2 - \pi_{key(S_1)} S_1)$ the set of tuples in S_2 not joining with any tuple in S_1. The *prioritized merge operator* is defined as follows:

$$S_1 \lhd S_2 = \Phi(S_1 \sqsupset\!\bowtie S_2, S_2) \cup (S_1 \bowtie\!\sqsubset S_2') \qquad \square$$

The prioritized merge operator includes all tuples of the left relation and only the tuples of the right relation whose key does not identify any tuple in the left relation. Moreover, only tuples 'coming' from the left relation are extended since tuples coming from the right relation, joining some tuples coming from the left relation, are not included. Thus, when integrating relations conflicting on the key attributes, the prioritized merge operator gives preference to the tuples of the left side relation and completes them with values taken from the right side relation.

Example 3. Consider the source relations S_1 and S_2 of Example 1. The relation $T = S_1 \triangleleft S_2$ is:

K	$Title$	$Author$	$Year$
1	$Moon$	$Greg$	$\perp$
2	$Money$	$Jones$	$\perp$
3	Sky	$Jones$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

T

The merged relation obtained in this case differs from the one of Example 1 because it does not contain the tuple $(3, Flowers, Smith, 1965)$ coming from relation S_2. □

Proposition 3. *Let S_1 and S_2 be two relations, then:*

- $S_1 \triangleleft S_2 \subseteq S_1 \boxtimes S_2$,
- $S_1 \boxtimes S_2 = (S_1 \triangleleft S_2) \cup (S_2 \triangleleft S_1)$,
- $\forall\, t \in inst(R_1 \triangleleft R_2)\ \exists t'$ *s.t.* ($t' \in R_1$ *or* $t' \in R_2$) *and* $t' \sqsubseteq t$ ($\sqsubseteq$*-Correctness),*
- $S_1 \triangleleft S_1 = S_1$. □

Obviously, the prioritized merge operator is not $\sqsubseteq$-Complete because some information contained in the right input relation might be lost. In fact, when there are tuples belonging to different input relations and having the same key values but different values for some other attributes, for these attributes the operator holds only the values coming from the left relation.

The prioritized merge operation $S_1 \triangleleft S_2$, introduced in Definition 6 can easily be expressed by means of an SQL statement, as follows:

```
SELECT S1.K, S1.B, COALESCE(S1.a1, S2.a1), .., COALESCE(S1.an, S2.an), S2.C
FROM   S1 LEFT OUTER JOIN S2 ON S1.K = S2.K
UNION
SELECT S2.K, NULL(B), S2.A, S2.C
FROM   S2, S1
WHERE S2.K NOT IN (SELECT S1.K FROM S1)
```

where the function `NULL`(B) assigns null values to the attributes in B.

4 Managing Inconsistent Data

We assume that each mediator component involved in the integration process contains an explicit representation of intentional knowledge, expressed by means of integrity constraints. Integrity constraints, usually defined by first order formulas or by means of special notations, express semantic information about data, i.e. relationships that must hold among data. Generally, a database D has a set of integrity constraints $\mathcal{IC}$ associated with it. D is said to be *consistent* if $D \models \mathcal{IC}$, otherwise it is inconsistent. In this paper we concentrate on functional dependencies. We present a technique, introduced in [13], which permits us to compute consistent answers for possibly inconsistent databases. The technique is based on the generation of a disjunctive program $\mathcal{DP}(\mathcal{IC})$ derived from the set of integrity constraints $\mathcal{IC}$.

4.1 Repairing and Querying Inconsistent Databases

Repaired databases are consistent databases which are derived from the source database by means of a minimal set of insertion and deletion of tuples. Given a (possibly inconsistent) database D, a *repair* for D is a pair of sets of atoms (R^+, R^-) such that *(i)* $R^+ \cap R^- = \emptyset$, *(ii)* $D \cup R^+ - R^- \models \mathcal{IC}$ and *(iii)* there is no pair $(S^+, S^-) \neq (R^+, R^-)$ such that $S^+ \subseteq R^+$, $S^- \subseteq R^-$ and $D \cup S^+ - S^- \models \mathcal{IC}$. The database $D \cup R^+ - R^-$ will be called the *repaired database*.

Thus, given a repair R, R^+ denotes the set of tuples which will be added to the database, whereas R^- denotes the set of tuples of D which will be deleted. In the following, for a given repair R and a database D, $R(D) = D \cup R^+ - R^-$ denotes the application of R to D.

It is possible to compute repairs for an inconsistent database by evaluating the extended Datalog program $\mathcal{DP}(\mathcal{IC})$, obtained by rewriting the integrity constraints $\mathcal{IC}$ into disjunctive rules, over the extensional database. In the rest of this section we focus our attention on functional dependencies, i.e. a particular class of integrity constraints.

Definition 7. Let c be a functional dependency $x \to y$ over P, which can be expressed by a formula of the form $(\forall x, y, z, u, v)[\, P(x, y, u) \wedge P(x, z, v) \supset y = z\,]$ then, $dj(c)$ denotes the extended disjunctive rule

$$\neg P(x, y, u) \vee \neg P(x, z, v) \leftarrow P(x, y, u),\ P(x, z, v), y \neq z$$

Let $\mathcal{IC}$ be a set of functional dependencies, then $\mathcal{DP}(\mathcal{IC}) \ = \{\ dj(c) \mid c \in \mathcal{IC}\ \}$. $\Box$

Thus, $\mathcal{DP}(\mathcal{IC})$ denotes the set of disjunctive rules obtained by rewriting $\mathcal{IC}$. $\mathrm{SM}(\mathcal{DP}(\mathcal{IC}), D)$ denotes the set of stable models of $\mathcal{DP}(\mathcal{IC}) \cup D$.

Given a database D and a set of functional dependencies IC defined on its schema, every stable model in $\mathrm{SM}(\mathcal{DP}(\mathcal{IC}), D)$ can be used to define a possible repair for the database by interpreting atoms with classical negation as deletions of tuples. In particular, if a stable model M contains two atoms $\neg\, p(t)$

(derived atom) and $p(t)$ (base atom) we deduce that the atom $p(t)$ violates some constraint and, therefore, it must be deleted.

The technique is sound and complete as each stable model defines a repair and each repair is derived from a stable model [13].

Consistent answers to queries against a possibly inconsistent database can be computed without modifying the database, on the basis of the stable models of the program corresponding to the integrity constraints and evaluated over the database.

Definition 8. Given a database D, a set of functional dependencies $\mathcal{IC}$ and a query $Q(g, \mathcal{P})$, the consistent answer of Q over D consists of the three distinct sets denoting, respectively, true, undefined and false atoms:

- $Ans^+(Q, D, \mathcal{IC}) = \{\ g(t) \in D \mid\ \nexists M \in \mathrm{SM}(\mathcal{DP}(\mathcal{IC}), D)\ s.t.\ \neg g(t) \in M\ \}$
- $Ans^u(Q, D, \mathcal{IC}) = \{\ g(t) \in D \mid \exists M_1, M_2 \in \mathrm{SM}(\mathcal{DP}(\mathcal{IC}), D)\ s.t.$
  $\neg g(t) \in M_1\ and\ \neg g(t) \notin M_2\ \}$
- $Ans^-(Q, D, \mathcal{IC})$ denotes the set of atoms which are neither true nor undefined (false atoms). □

The consistent answer to a query $Q = (g, \mathcal{P})$ against a database D under the certain semantics is given by $Ans^+(Q, D, \mathcal{IC})$. Analogously, the answer to $Q = (g, \mathcal{P})$ under the possible semantics constists in

$$Ans^+(Q, D, \mathcal{IC}) \bigcup Ans^u(Q, D, \mathcal{IC}).$$

Observe that, true atoms appear in all repaired databases whereas undefined atoms appear in a proper subset of repaired databases.

Theorem 1. *Let D be an integrated database, $\mathcal{IC}$ be a set of functional dependencies and Q be a query, then the computation of a consistent answer of Q over D can be done in polynomial time.* □

Example 4. Consider the integrated relation T of Example 1 and the functional dependency $K \to (Title, Author, Year)$ stating that K is a key for the relation. The functional dependency can be rewritten as first order formulas:

$$(\forall x, y, z, w, y', z', w')\ [T(x, y, z, w), T(x, y', z', w') \supseteq y = y', z = z', w = w']$$

The associated disjunctive program is

$$\neg T(x, y, z, w) \vee \neg T(x, y', z', w') \leftarrow T(x, y, z, w), T(x, a, b, c), (y \neq a \vee z \neq b \vee w \neq c)$$

The above program has two stable models $M_1 = D \cup \{\neg T(3, Sky, Jones, 1965)\}$ and $M_2 = D \cup \{\neg T(3, Flowers, Smith, 1965)\}$. Thus, the repairs for T are $R_1 = (\{T(3, Flowers, Smith, 1965)\}, \emptyset)$ and $R_2 = (\{T(3, Sky, Jones, 1965)\}, \emptyset)$ producing, respectively, the repaired databases $R_1(T)$ and $R_2(T)$ reported in Figure 2.

Finally, the answer to the query asking for the title of the book with code 2 is $Money$ whereas the answer to the query asking for the title of the book with code 3 is unknown since there are two alternative values. □

K	$Title$	$Author$	$Year$
1	$Moon$	$Greg$	$\perp$
2	$Money$	$Jones$	$\perp$
3	$Flowers$	$Smith$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

$R_1(T)$

K	$Title$	$Author$	$Year$
1	$Moon$	$Greg$	$\perp$
2	$Money$	$Jones$	$\perp$
3	Sky	$Jones$	1965
4	Sea	$Taylor$	1971
7	Sun	$Steven$	1980

$R_2(T)$

Fig. 2.

References

1. S. Argaval, A.M. Keller, G.Wiederhold, and K. Saraswat. Flexible Relation: an Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In IEEE *Int. Conf. on Data Engineering*,1995.
2. M. Arenas, L. Bertossi, J. Chomicki, Consistent Query Answers in Inconsistent Databases. *Proc. PODS 1999*, pp. 68–79, 1999.
3. C. Baral, S. Kraus, J. Minker, Combining Multiple Knowledge Bases. *IEEE-Trans. on Knowledge and Data Engineering*, 3(2): 208-220 (1991)
4. Y. Breitbart, Multidatabase interoperability. *Sigmod Record 19(3)* (1990), 53–60.
5. F. Bry, Query Answering in Information System with Integrity Constraints, In *IFIP WG 11.5 Working Conf. on Integrity and Control in Inform. System*, 1997.
6. P. M. Dung, Integrating Data from Possibly Inconsistent Databases. *Proc. Int. Conf. on Cooperative Information Systems*, 1996: 58-65
7. H. Garcia-Molina, Y. Papakonstantinou, D. Quass,A. Rajaraman , Y. Sagiv, J. Ullman, V. Vassalos, J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems 8* (1997), 117–132.
8. Gelfond, M., Lifschitz, V. The Stable Model Semantics for Logic Programming, *ICLP Conf.* pages 1070–1080, 1988.
9. M. Gelfond,V. Lifschitz(1991), Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing*, *9*, 365–385.
10. Greco, S., and Saccà, D., Negative Logic Programs. *North American Conf. on Logic Progr.*, pages 480-497, 1990.
11. Greco, S., and Zumpano E., Querying Inconsistent Database *LPAR Conf.*, pages 308-325, 2000.
12. J. Grant, V. S. Subrahmanian: Reasoning in Inconsistent Knowledge Bases. *IEEE-Trans. on Knowledge and Data Eng.*, 7(1): 177-189, 1995
13. S. Greco, E. Zumpano Querying Inconsistent Database *Logic for Programming and Automated Reasoning*, pages 308-325, 2000.
14. R. Hull, Managing Semantic Heterogeneity in Databases: a Theoretical Perspective, *Proc. Symposium on Principles of Database Systems*, 1997: 51-61
15. Kowalski, R. A., and Sadri, F., Logic Programs with Exceptions. *NGC Jou.*, 9(No. 3/4), 387-400, 1991.
16. A. Levy, A. Rajaraman, J. Ordille, Querying heterogeneous information sources using source descriptions. *Proc. VLDB Conf.*, 1996, pages 251–262.
17. J. Lin, A. O. Mendelzon, Knowledge Base Merging by Majority, in R. Pareschi and B. Fronhoefer (eds.), *Dynamic Worlds*, Kluwer, 1999. Kluwer, 1999.
18. Minker, J. On Indefinite Data Bases and the Closed World Assumption, *6-th Conf. on Automated Deduction*, pages 292–308, 1982.
19. J.K. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, Rockville, Md., 1988.

20. J. D. Ullman, Information Integration Using Logical Views, *Theoretical Computer Science*, 239(2) 2000: 189-210
21. L.L. Yan, M. T. Ozsu, Conflict Tolerant Queries in Aurora *Proc. Coopis Conf.*, 1999: 279-290
22. G. Wiederhold, Mediators in the architecture of future information systems. *IEEE Computer 25(3)* (1992), 38–49.

A Knowledge Engineering Approach to Deal with ‘Narrative’ Multimedia Documents

Gian Piero Zarri

Centre National de la Recherche Scientifique (CNRS),
44 rue de l’Amiral Mouchez,
F-75014 Paris, France
zarri@ivry.cnrs.fr

Abstract. We describe here the last developments about NKRL (Narrative Knowledge Representation Language), a conceptual modeling formalism used to deal with ‘narrative’ multimedia documents. Narrative documents of an industrial and economic interest correspond, e.g., to news stories, corporate documents, normative and legal texts, intelligence messages and medical records. A new Java version of NKRL, RDF- and XML-compliant, has been recently built-up in the framework of the CONCERTO Esprit project.

1 Introduction

Narrative documents, or ‘narratives’, are multimedia documents that describe the actual (or intended) state or behavior of some ‘actors’ (or ‘characters’, ‘personages’ etc.). These try to attain a specific result, experience particular situations, manipulate some (concrete or abstract) materials, communicate with other people, send or receive messages, buy, sell, deliver etc. Leaving pure fiction aside, we can note that:

- A considerable amount of the natural language (NL) information that is relevant from an economic point of view deals with narratives. This is true for all the different sorts of news story documents, for most of corporate information (memos, policy statements, reports, minutes etc.), for many intelligence messages, for medical records, notarized deeds, sentences and other legal documents, etc.
- In the narrative documents, the actors or personages are not necessarily human beings. We can have narrative documents concerning, e.g., the vicissitudes in the journey of a nuclear submarine (the ‘actor’, ‘subject’ or ‘character’) or the various avatars in the life of a commercial product. This personification process can be executed to a very large extent, giving then rise to narrative documents very removed from a pure human context.
- It is not even necessary that the narrative situations to deal with be recorded in NL documents. Let us consider a collection of Web images, where one represents an information that, verbalized, could be expressed as “Three nice girls are lying on the beach”. Having at our disposals tools — like those described in this paper — for

coding the ‘meaning’ of narrative documents in a machine-understandable way, we can directly ‘annotate’ the picture without any recourse to a previous rendering in natural language. The same is, obviously, possible for ‘narrative’ situations described in video or digital audio documents.

Being able to represent the semantic content of narratives — i.e., their key ‘meaning’ — in a general, accurate, and effective way is then both conceptually relevant and economically interesting.

In this paper, we will describe the last developments concerning the use of NKRL (Narrative Knowledge Representation Language), see [1, 2, 3, 4], to represent the gist of economically relevant narratives. Note that NKRL has been used as ‘the’ modeling language for narratives in European projects like Nomos (Esprit P5330), Cobalt (LRE P61011) WebLearning (GALILEO Actions), and in the recent Concerto project (Esprit P29159). In Concerto, NKRL was used to encode the ‘conceptual annotations’ (formal description of the main semantic content of a document) that must added to digital documents to facilitate their ‘intelligent’ retrieval, processing, displaying, etc. [5]. NKRL is now employed in the new Euforbia project (IAP P26505) to encode the rules used for filtering Internet documents according to a semantic-rich approach.

2 General Information about NKRL

Traditionally, the NKRL knowledge representation tools are presented as organized into four connected ‘components’, the definitional, enumerative, descriptive and factual component.

The definitional component supplies the tools for representing the ‘concepts’, intended here as the important notions that it is absolutely necessary to take into account in a given application domain. In NKRL, a concept is represented, substantially, as a frame-like data structure, i.e., as an n-ary association of triples ‘name-attribute-value’ that have a common ‘name’ element. This name corresponds to a symbolic label like *human_being*, *taxi_* (the general class referring to all the possible taxis, not a specific cab), *city_*, *chair_*, *gold_*, etc. NKRL concepts are inserted into a generalization/specialization hierarchy that, for historical reasons, is called H_CLASS(es), and which corresponds well to the usual ontologies of terms. The minimal requirements concerning a frame-based language suitable for playing the role of ‘definitional component’ in NKRL are expressed, e.g., in [1]. We will not dwell here on the H_CLASS (definitional component) characteristics given that the most ‘interesting’ reasoning mechanisms of NKRL are associated with the descriptive and factual components, see below. The most used H_CLASS inferential mechanism is based simply on usual the generic/specific (subsumption) relationship among concepts, systematically employed, e.g., in all the NKRL applications for checking the constraints on the variables, see below.

A fundamental assumption about the organization of the hierarchy of concepts in NKRL concerns the differentiation between ‘notions which can be instantiated directly into enumerable specimens', like "chair" (a physical object) and 'notions which cannot be instantiated directly into specimens', like "gold" (a substance). The two high-level branches of H_CLASS take origin, therefore, from two concepts labeled as *sortal_concepts* and *non_sortal_concepts*. The specializations of the former, like *chair_*, *city_* or *european_city* can have direct instances (chair_27, paris_), whereas the specializations of the latter, like *gold_* or *colour_*, admit further specializations, see *white_gold* or *red_*, but do not have direct instances.

The 'enumerative component' of NKRL concerns the tools for the formal representation, as (at least partially) instantiated frames, of the concrete realizations (lucy_, taxi_53, chair_27, paris_) of the H_CLASS sortal concepts. In NKRL, the instances of sortal concepts take the name of *individuals*. Individuals are then countable and, like the concepts, possess unique symbolic labels (lucy_ etc.). Throughout this paper, I will use the italic type style to represent a *concept_*, the roman style to represent an individual_. Individuals, unlike concepts, are always associated with spatio-temporal co-ordinates that can, sometimes, refer to very hypothetical worlds, see, e.g., the individual green_firespitting_dragon_49. Note also that individuals do not have a proper hierarchical organization — instances of individuals are not allowed in NKRL, see [1]. They are nevertheless associated with the H_CLASS hierarchy, given that they appear as the 'leaves' of this hierarchy, directly linked with the sortal concepts. Assuming then the existence, in a particular NKRL application, of a database of individuals, this last can be considered as indexed by the H_CLASS hierarchical structure.

The 'descriptive' and 'factual' tools concern the representation of the 'events' proper to a given domain — i.e., the coding of the *interactions* among the *particular concepts and individuals* that *play a role* in the contest of these events.

The descriptive component includes the modeling tools used to produce the formal representations (called 'templates') of some *general narrative classes*, like "moving a generic object", "formulate a need", "having a negative attitude towards someone", "be present somewhere". In contrast to the traditional frame structures used for concepts and individuals, templates are characterized by the association of quadruples connecting together the *symbolic name* of the template, a *predicate* and the *arguments* of the predicate introduced by named relations, the *roles*. The quadruples have in common the 'name' and 'predicate' elements. If we denote then with L_i the generic symbolic label identifying a given template, with P_j the predicate used in the template, with R_k the generic role and with a_k the corresponding argument, the NKRL data structures for templates have the following general format:

$$(L_i\ (P_j\ (R_1\ a_1)\ (R_2\ a_2)\ \ldots\ (R_n\ a_n)))\ , \qquad \textbf{(1)}$$

see the example in Figure 1, commented below. Presently, the predicates pertain to the set {BEHAVE, EXIST, EXPERIENCE, MOVE, OWN, PRODUCE, RECEIVE}, and the roles to the set {SUBJ(ect), OBJ(ect), SOURCE, BEN(e)F(iciary), MODAL(ity), TOPIC, CONTEXT}.

Templates are structured into an inheritance hierarchy, H_TEMP(lates), which corresponds, therefore, to a new sort of ontology, an ‘ontology of events’.

The instances (called ‘predicative occurrences’) of the templates, i.e., the representation of single, specific elementary events — see examples like “Tomorrow, I will move the wardrobe” or “Lucy was looking for a taxi” — are, eventually, in the domain of the last component, the factual one. Predicative occurrences are always associated with the indication of the temporal interval where the corresponding elementary event ‘holds’. The interval is delimited by two ‘timestamps’ $<t_1, t_2>$ such that $t_1 \leq t_2$ — the meaning of ‘$\leq$’ is here intuitive. A detailed description of the ‘theory’ of temporal representation in NKRL can be found in [2].

3 A Simple Example

To represent an elementary event like “On April 5th, 1982, Francis Pym is appointed Foreign Secretary by Margaret Thatcher” under the form of predicative occurrence (factual component), we must select firstly the template corresponding to ‘nominate to a post’, which is represented in the upper part of Figure 1.

The ‘position’ code shows the place of this ‘nomination’ template within the OWN branch (5.) of the H_TEMP hierarchy, see also Figure 2: this template is then a specialization (see the ‘father’ code) of the particular `OWN` template of H_TEMP that corresponds to ‘being in possession of a post’. The (mandatory) presence of a ‘temporal modulator’, ‘begin’ indicates that the only timestamp (t_1) which can be associated with the predicative occurrences derived from the ‘nomination’ template corresponds to the beginning of the state of being in possession — here, to the nomination. In the occurrences, time stamps are represented through two ‘temporal attributes’, `date-1` and `date-2`, see [2]. In the occurrence `c1` of Figure 1, this interval is reduced to a point on the time axis, as indicated by the single value, ‘`5-april-1982`’ (the nomination date), associated with the attribute `date-1`.

The argument of the predicate (the a_k terms in formula (**1**) of the previous Section) are represented by variables with associated constraints; these last are expressed as concepts or combinations of concepts, i.e., making use of the terms of the H_CLASS hierarchy (definitional component). The arguments associated with the `SUBJ(ect), OBJ(ect), SOURCE` and `BEN(e)F(iciary)` roles can be further specified by denoting the ‘place’ (location) they occupy within the framework of a particular event. For example, we could add that, at the time of the nomination, both Francis Pym and Margaret Thatcher were in London. These ‘location attributes’ — represented by the variables *`var2`* and *`var5`* in the template of Figure 1, and implemented as lists in the predicative occurrences, see, e.g., occurrence `c3` in Figure 3 below — are linked with the predicate arguments by using the colon operator, ‘:’. The constituents (as SOURCE in Figure 1) included in square brackets are optional; the symbol $\{\}^*$ means that a constituent is forbidden for a given template.

The role fillers in the predicative occurrence represented in the lower part of Figure 1 conform to the constraints of the father-template. For example, `francis_pym` is an individual (enumerative component) instance of the sortal concept *individual_person* that is, in turn, a specialization of *human_being*; *foreign_secretary* is a specialization of *post_*, etc. — note that the filler of a SOURCE role always represents the ‘originating factor’ of the event.

```
name: Own:NamingToPost
father: Own:BeingInPossessionOfPost
position: 5.1221
NL description: ‘A Human Being is Appointed to a Post’

OWN  SUBJ         var1: [(var2)]
     OBJ          var3
     [SOURCE      var4: [(var5)]]
     {BENF}*
     [MODAL       var6]
     [TOPIC       var7]
     [CONTEXT     var8]
     { [ modulators ], begin }

     var1   =     <human_being>
     var3   =     <post_>
     var4   =     <human_being_or_social_body>
     var6   =     <action_name>
     var7   =     <property_>
     var8   =     <event_> | <action_name>
     var2, var5 = <physical_location>


c1)   OWN    SUBJ    francis_pym
             OBJ     foreign_secretary
             SOURCE  margaret_thatcher
             [begin]
             date-1: (5-april-1982)
             date-2:
```

Fig. 1. Deriving a predicative occurrence from a template

We can note an important point. Unlike, e.g., canonical graphs in Sowa’s conceptual graphs theory, see [6], which must be explicitly defined for each new application, the (about 200) templates that make up actually the H_TEMP hierarchy are fixed and fully defined — H_TEMP corresponds then to a sort of ‘catalogue’ of NKRL templates, and we can say that these templates are part and parcel of the definition of the language. An (extremely simplified) image of the present H_TEMP hierarchy is given in Figure 2. Note that, when needed, it is easy to derive from the existing templates new templates that it could be required for a particular application. H_TEMP is then a growing structure. On the other hand, H_TEMP can be customized by making use of a ‘sub-catalogue’ including only the templates strictly relevant to a given domain.

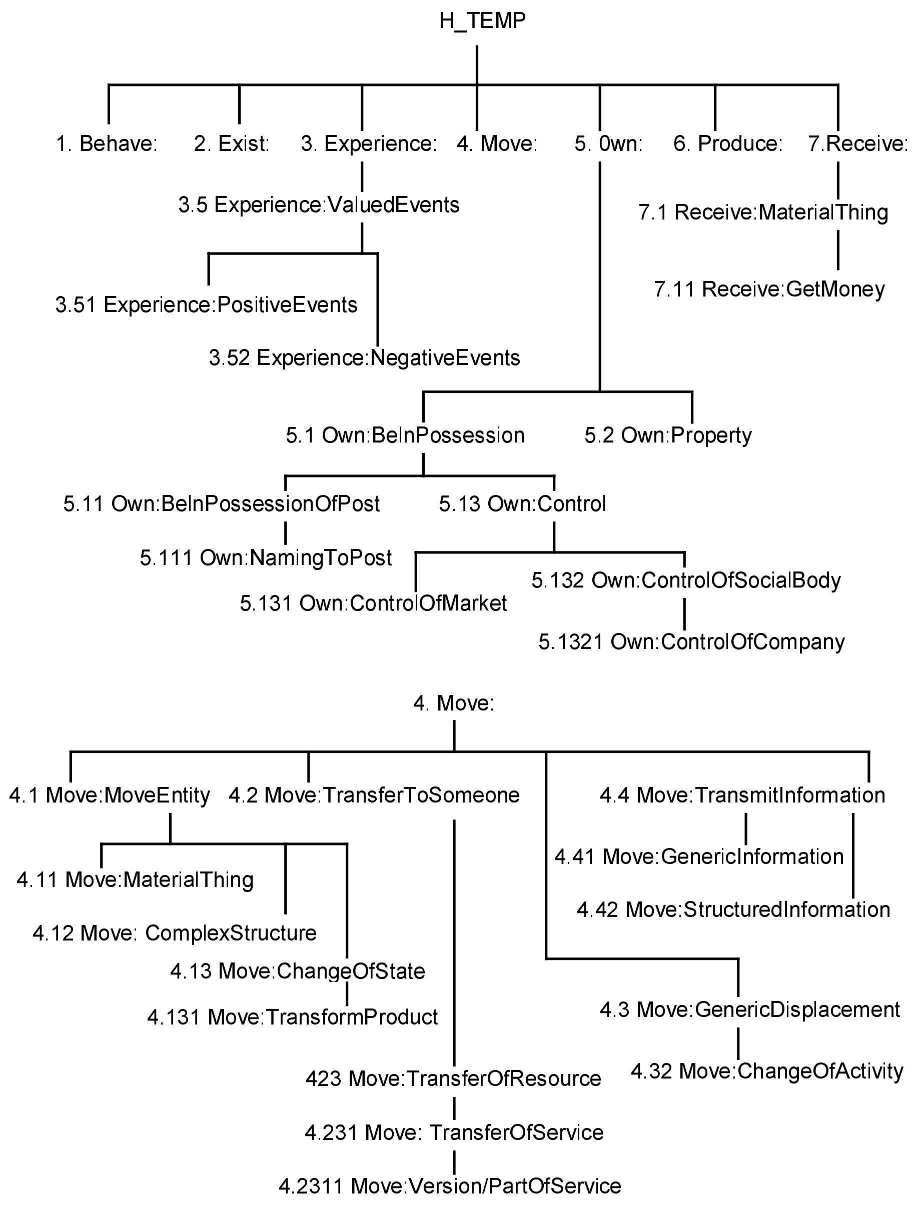

H_TEMP
1. Behave:
2. Exist:
3. Experience:
4. Move:
5. 0wn:
6. Produce:
7.Receive:
3.5 Experience:ValuedEvents
3.51 Experience:PositiveEvents
3.52 Experience:NegativeEvents
7.1 Receive:MaterialThing
7.11 Receive:GetMoney
5.1 Own:BeInPossession
5.2 Own:Property
5.11 Own:BeInPossessionOfPost
5.13 Own:Control
5.111 Own:NamingToPost
5.131 Own:ControlOfMarket
5.132 Own:ControlOfSocialBody
5.1321 Own:ControlOfCompany
4. Move:
4.1 Move:MoveEntity
4.2 Move:TransferToSomeone
4.4 Move:TransmitInformation
4.11 Move:MaterialThing
4.12 Move: ComplexStructure
4.13 Move:ChangeOfState
4.131 Move:TransformProduct
4.41 Move:GenericInformation
4.42 Move:StructuredInformation
4.3 Move:GenericDisplacement
4.32 Move:ChangeOfActivity
423 Move:TransferOfResource
4.231 Move: TransferOfService
4.2311 Move:Version/PartOfService

Fig. 2. A (simplified) picture of the H_TEMP hierarchy (descriptive component)

This approach is particularly advantageous for practical applications, and it implies, in particular, that: i) a system-builder does not have to create himself the structural knowledge needed to describe the events proper to a (sufficiently) large class of narrative documents; ii) it becomes easier to secure the reproduction or the sharing of previous results. Expressed in different terms, we can say that the NKRL approach implies the addition of an 'ontology of events', H_TEMP, to the traditional 'ontology of concepts', H_CLASS, augmenting then greatly the global interoperability of the language.

4 Additional Properties of the NKRL Language

The basic NKRL tools are enhanced by two additional classes of conceptual structures:

- the AECS 'sub-language', see [7], that allows the construction of complex (structured) predicate arguments, called 'expansions';
- the second order tools (binding structures and completive construction), see [2], used to Lcode the 'connectivity phenomena' (logico-semantic links) that, in a narrative situation, can exist between single narrative fragments (corresponding to single NKRL predicative structures).

We will introduce informally the two new tools making use of additional examples of NKRL coding of narrative structures. Figure 3 translates then this fragment of news story: "This morning, the spokesman said in a newspaper interview that, yesterday, his company has bought three factories abroad". `today_` and `yesterday_` are two fictitious individuals introduced here, for simplicity's sake, in place of real or approximate dates, see [2]. Note that the arguments of the predicate, and the templates/occurrences as a whole, may be characterized by the presence of particular codes, determiners or attributes (like the 'temporal modulators' introduced in the previous Section), which give further details about their significant aspects. For example, as already noticed, the location attributes, represented as lists, are linked with the arguments by using the colon operator, see `c3` in Figure 3 (*`abroad_`*).

The 'attributive operator', `SPECIF(ication)`, which appears in both the occurrences `c2` and c3, is one of the four operators that make up the AECS sub-language. AECS includes the disjunctive operator (`ALTERNative` = A), the distributive operator (`ENUMeration` = E), the collective operator (`COORDination` = C), and the attributive operator (`SPECIFication` = S).

The meaning of `ALTERN(ative)` is self-evident. Informally, the semantics of `SPECIF(ication)` can be explained in this way: the `SPECIF` lists, with syntax (`SPECIF` e_i p_1 … p_n), are used to represent some of the properties p_i that can be asserted about the first argument e_i, concept or individual, of the operator, e.g., `human_being_`1 and *`spokesman_`* in the occurrence `c2` of Figure 3.

In a COORD(ination) list, all the elements of the expansion take part — *necessarily together* — in the particular relationship with the predicate defined by the role to be filled; in an ENUM(eration) framework, this simultaneity is not required. As an example, we can imagine a situation where the spokesman has communicated his information to two different newspapers, newspaper_1 and newspaper_2, the BEN(e)F(iciaries) — BENF is the role to be filled. If the two newspapers have assisted *together* to the interview, i.e., if they have received the information *together*, then the BENF slot of c2 will be filled with (COORD newspaper_1 newspaper_2). On the contrary, if the information was received *separately* — which can correspond, in practice, to a situation where the newspapers have taken part in two different interviews — then the BENF filler becomes: (ENUM newspaper_1 newspaper_2). ENUM can then be considered as a shortcut for reducing the size of a database of predicative occurrences by merging quite similar occurrences. For more information about the AECS operators and, in particular, about the rules for mixing them within the same expansion ('priority rule'), see [2].

```
c2) MOVE SUBJ    (SPECIF human_being_1
                          (SPECIF spokesman_ company_1))
         OBJ     #c3
         BENF    newspaper_1
         MODAL   interview_
         date-1: today_
         date-2:

c3) PRODUCE SUBJ    company_1
            OBJ     (SPECIF purchase_1 (SPECIF factory_99
                          (SPECIF cardinality_ 3))): (abroad_)
            date-1: yesterday_
            date-2:

     [ factory_99
              InstanceOf : factory_
              HasMember   : 3 ]
```

Fig. 3. An example of completive construction

The last item of Figure 3 supplies an example of enumerative data structure, explicitly associated with the individual factory_99 according to the rules for coding 'plural situations' in NKRL, see [1]. The non-empty HasMember slot in this structure makes it clear that the individual factory_99, as mentioned in c3, is referring in reality to several instances of *factory_*. Individuals like factory_99 are 'collections' rather then 'sets' (all the NKRL concepts can be interpreted as sets), given that the extensionality axiom (two sets are equal iff they have the same elements) does not hold here. In our framework, two collections, say factory_99 and factory_100, can be co-extensional, i.e., they can include exactly the same elements, without being necessarily considered as identical if created at different moments in time in the context of totally different event, see [8]. In Figure 3, we have supposed that the three factories were, *a priori*, not sufficiently important in the context of the news story to justify their explicit representation as specific individuals, e.g., `factory_1`, `factory_2`, `factory_3`. Note that, if not expressly required by the characteristics of the application, a basic NKRL principle suggests that we should try to avoid any unnecessary proliferation of individuals.

In coding narrative information, one of the most difficult problems consists in being able to deal (at least partly) with the 'connectivity phenomena' like causality, goal, indirect speech, co-ordination and subordination, etc. — in short, all those phenomena that, in a sequence of statements, cause the global meaning to go beyond the simple addition of the information conveyed by each single statement. In NKRL, the connectivity phenomena are dealt with making a (limited) use of second order structures: these are obtained from a *reification* of the predicative occurrences that is based on the use of their symbolic labels — like `c2` and `c3` in Figure 3. A first example of second order structure is given by the so-called 'completive construction', that consists in accepting as filler of a role in a predicative occurrence the symbolic label of another predicative occurrence. For example, see Figure 3, we can remark that the basic, `MOVE` template (descriptive component) that is at the origin of `c2` is systematically used to translate any sort of explicit or implicit transmission of an information ("The spokesman said…"). In this example of completive construction, the filler of the `OBJ(ect)` slot in the occurrence (`c2`) which materializes the 'transmission' template is a symbolic label (`c3`) that refers to another occurrence, i.e. the occurrence bearing the informational content to be spread out (" …the company has bought three factories abroad").

Figure 4 corresponds now to a narrative information that can be rendered in natural language as: "We notice today, 10 June 1998, that British Telecom will offer its customers a pay-as-you-go (payg) Internet service".

```
c4)  (GOAL  c5  c6)

c5)  BEHAVE  SUBJ   british_telecom
     { obs }
     date1:  10-june-1998
     date2:

*c6) MOVE    SUBJ    british_telecom
             OBJ     payg_internet_service_1
             BENF    (SPECIF customer_ british_telecom)
             date1:  after-10-june-1998
             date2:
```

Fig. 4. An example of binding occurrence

To translate the general idea of 'acting to obtain a given result', we use then:

- A predicative occurrence (`c5` in Figure 4), instance of a basic template pertaining to the `BEHAVE` branch of the template hierarchy (H_TEMP), and corresponding to the general meaning of 'focusing on a result'. This occurrence is used to express the 'acting' component, i.e., it allows us to identify the `SUBJ(ect)` of the action, the temporal co-ordinates, possibly the `MODAL(ity)` or the instigator (`SOURCE`), etc.
- A second predicative occurrence, `c6` in Figure 4, with a different NKRL predicate and which is used to express the 'intended result' component. This second occurrence, which happens 'in the future' with respect to the previous one (`BEHAVE`), is marked as hypothetical, i.e., it is always characterized by the presence of an uncertainty validity attribute, code '`*`'. Expressions like `after-10-june-1998` are concretely rendered as date ranges, see [2].
- A 'binding occurrence', c4 in Figure 4, linking together the previous predicative occurrences and labeled with `GOAL`, an operator pertaining to the taxonomy of causality of NKRL, see [2].

Binding structures — i.e., lists where the elements are symbolic labels, `c5` and `c6` in Figure 4 — are another example of second-order structures used to represent the connectivity phenomena. The general schema for representing the 'focusing on an intended result' domain in NKRL is then:

c_α) (GOAL c_β c_γ)
c_β) BEHAVE SUBJ <*human_being_or_social_body*>
*c_γ) <`predicative_occurrence`, with any syntax> .

In Figure 4 '`obs(erve)`' is, like '`begin`' and '`end`', a temporal modulator, see [2]. '`obs`' is used to assert that the event related in the occurrence 'holds' at the date associated with `date-1` without, at this level, giving any detailed information about the beginning or end of this event, which normally extends beyond the given date. Note that the addition of a '`ment(al)`' modulator in the `BEHAVE` occurrence, c_β, which introduces an 'acting to obtain a result' construction should imply that no concrete initiative is taken by the `SUBJ` of `BEHAVE` in order to fulfil the result. In this case, the 'result', *c_γ, reflects only the wishes and desires of the `SUBJ(ect)`.

5 Implementation Notes

A new version of NKRL, implemented in Java and XML/RDF compliant, has been realized in the CONCERTO's framework. We recall here that the RDF model, see [9, 10], is a framework for describing general-purpose metadata that is promoted by the W3C Committee and is based on XML. In knowledge representation terms, the basic RDF data structure can be assimilated to a *triple* where two 'resources' are linked by a 'property' that behaves like a *dyadic* conceptual relation. A description of some problems encountered when translating into RDF the NKRL data structures, and of the solutions adopted in this context, can be found in [3, 4]. These problems have concerned mainly i) the correct representation of the NKRL variables and of their constraints; ii) the rendering in RDF of the *four* AECS operators making use only of *three* RDF 'containers', 'Bag', 'Sequence' and 'Alternative'. In Figure 5, we reproduce the native NKRL coding (above) and the NKRL/RDF CONCERTO coding (below) of an extremely simple narrative fragment: "On June 12, 1997, John and Peter were admitted (*together*) to hospital" — note that adding the indication 'together' forces an interpretation in a COORD style. Note also that, in the RDF coding, the COORD list that represents the filler of the SUBJ role in NKRL is rendered as a Bag, see [9].

```
c7)  EXIST  SUBJ   (COORD john_ peter_): (hospital_1)
            { begin }
            date-1: 2-june-1997
            date-2:

<?xml version="1.0" ?>
<!DOCTYPE DOCUMENTS SYSTEM "CA_RDF.dtd">
<CONCEPTUAL_ANNOTATION>
  <rdf:RDF xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
  xmlns:rdf=”http://www.w3.org/TR/1999/PR-rdf-schema-19990303#”
  xmlns:ca="http://projects.pira.co.uk/concerto#">
    <rdf:Description about="occ7">
    <rdf:type resource=”ca:Occurrence”/>
       <ca:instanceOf>Template2.31</ca:instanceOf>
       <ca:predicateName>Exist</ca:predicateName>
       <ca:subject rdf:ID=”Subj2.31” rdf:parseType=”Resource”>
       <concerto:filler>
         <rdf:Bag>
           <rdf :li rdf:resource="#john_"/>
           <rdf :li rdf:resource="#peter_"/>
         </rdf:Bag>
       </concerto:filler>
       <concerto:location>hospital_1</concerto:location>
       </ca:subject>
       <ca:listOfModulators>
          <rdf:Seq><rdf:li>begin</rdf:li></rdf:Seq>
      </ca:listOfModulators>
      <ca:date1>02/06/1997</ca:date1>
    </rdf:Description>
  </rdf:RDF>
</CONCEPTUAL_ANNOTATION>
```

Fig. 5. RDF/XML representation of a simple NKRL predicative occurrence

To supply now some details about the query and inferencing operations proper to NKRL, let us introduce informally the concept of ‘search pattern’. Search patterns — i.e., the formal counterparts of natural language queries — are NKRL data structures that correspond to partially instantiated templates and that supply the general framework of information to be searched for, by filtering or unification, within an NKRL knowledge base. A simple example of search pattern, translating the query: “Was John at the hospital in July/August 1997?” (see the upper part of Figure 5) is represented in Figure 6. The two timestamps associated with the pattern constitute now the ‘search interval’ that is used to limit the search for unification to the slice of time that it is considered appropriate to explore. In our example, the search pattern of Figure 6 can successfully unify occurrence c7 in Figure 5: in the absence of explicit, negative evidence, a given situation is assumed to persist within the immediate temporal environment of the originating event, see [2].

```
(?w  IS-PRED-OCCURRENCE
     :predicate         EXIST
     :SUBJ              john_
     :location of SUBJ  hospital_
     (1-july-1997, 31-august-1997))
```

Fig. 6. A search pattern that unifies the predicative occurrence of Figure 5

In the CONCERTO version of NKRL, a Java module called FUM module (Filtering Unification Module) deals with search patterns. Unification is executed taking into account, amongst other things, the fact that a ‘generic concept’ included in the search pattern can unify one of its ‘specific concepts’ — or the instances (individuals) of a specific concept — included in a corresponding position of the occurrence. ‘Generic’ and ‘specific’ refer, obviously, to the structure of the NKRL concept ontology, i.e., H_CLASS. Therefore, a search pattern asking which company has proposed a pay-as-you-go (payg) Internet service to his customers will retrieve the predicative occurrence `c6` of Figure 4, given that the `SUBJ` filler of the pattern, `company_`, subsumes in H_CLASS *`telecom_company`*, and the individual `british_telecom` in Figure 4 is an instance of *`telecom_company`*. From an algorithmic point of view, the main interest of FUM lies in the operations needed to unify correctly the complex fillers built up making use of the four AECS operators. These imply the decomposition of the complex fillers into tree structures labeled with the operators, and in the unification of the tree structures of the pattern with those of the occurrences following the priorities defined by the ‘priority rule’ already mentioned, see [2, 7].

The inference level supplied by FUM is only a first step towards more complex reasoning strategies. For example, the use of the ‘transformation rules’ allows to deal with the problem of obtaining a plausible answer from a database of predicative occurrences also in the absence of the explicitly requested information, by searching *semantic affinities* between what is requested and what is really present in the repository.

The principle employed consists in using these rules to automatically 'transform' the original query (i.e., the original search pattern) into one or more different queries (search patterns) that are not strictly 'equivalent' but only 'semantically close' to the original one, see [11].

We will now supply some information about another category of NKRL inference rules, the 'hypotheses', see the example of Figure 7.

Premise :

```
RECEIVE       SUBJ          x
              OBJ           money_
              SOURCE        y

x = company_
y = human_being | company_
```

"A company has received some money from another company or a physical person"

First condition schema :

```
PRODUCE       SUBJ          (COORD x y)
              OBJ           z
              BENF          (COORD x y)
              TOPIC         (SPECIF process_ v)

z = mutual_commitment | business_agreement
v = tool/product_
```

"A general of business-oriented agreement about the creation of a new product has been concluded by the two parties mentioned in the premise"

Second condition schema :

```
PRODUCE       SUBJ          x
              OBJ           v
              MODAL         w
              CONTEXT       z

w = industrial_process |technological_process
```

"The company that received the money has actually created the product mentioned in the first condition schema"

Fig. 7. An example of hypothesis rule

We suppose here to have directly retrieved, thanks to FUM, a given information within a base of NKRL occurrences, e.g., the information: “Pharmacopeia, an USA biotechnology company, has received 64,000,000 dollars from the German company Schering in connection with an R&D activity”. We suppose, moreover, that this occurrence is not already explicitly related with other occurrences of the base. Under these conditions, we can activate a specific Java module, InferenceEngine that, using a rule like that of Figure 7, will try to rely automatically the information found originally by FUM with other information in the base. If this is possible, this last information will represent a sort of ‘causal explanation’ of the information originally retrieved — in our example, an ‘explanation’ of the money paid by Schering. In our example, we will find, e.g., that “Pharmacopeia and Schering have signed an agreement concerning the production by Pharmacopeia of a new compound” and that “In the framework of the agreement previously mentioned, Pharmacopeia has actually produced the new compound”.

6 Conclusion

In this paper, we have described the last developments of NKRL (Narrative Knowledge Representation Language), a conceptual modeling formalism used for taking into account the characteristics of narrative documents. In these documents, the main part of the information content consists in the description of ‘events’ that relate the real or intended behavior of some ‘actors’ (characters, personages, etc.). Narrative documents of an industrial and economic interest correspond, for example, to news stories, corporate documents (memos, policy statements, reports and minutes), normative and legal texts, intelligence messages, representation of the patient's medical records, etc. NKRL makes use of several representational principles (concepts under the form of frames, templates, second order binding structures etc.) and of several high-level inference tools. Some information on the implementation aspects (and on the inference techniques) has also been supplied in the paper.

References

1. Zarri, G.P., NKRL, a Knowledge Representation Tool for Encoding the ‘Meaning’ of Complex Narrative Texts, Natural Language Engineering - Special Issue on Knowledge Representation for NL Processing in Implemented Systems, 3 (1997) 231-253.
2. Zarri, G.P., Representation of Temporal Knowledge in Events: The Formalism, and Its Potential for Legal Narratives, Information & Communications Technology Law - Special Issue on Models of Time, Action, and Situations, 7 (1998) 213-241.
3. Zarri, G.P., Metadata, a ‘Semantic’ Approach. In: Bench-Capon, T., Soda, G., Min Tjoa, A. (eds.): Database and Expert Systems Applications — Proceedings of the 10th International Conference, DEXA’99. Lectures Notes in Computer Science 1677. Springer-Verlag, Berlin (1999).

4. Zarri, G.P., A Conceptual Model for Capturing and Reusing Knowledge in Business-Oriented Domains. In: Roy, R. (ed.): Industrial Knowledge Management: A Micro-level Approach. Springer-Verlag, London (2000).
5. Armani, B., Bertino, E., Catania, B., Laradi, D., Marin, B., and Zarri, G.P., Repository Management in an Intelligent Indexing Approach for Multimedia Digital Libraries. In: Ras, Z.W., and Ohsuga, S. (eds.): Foundations of Intelligent Systems - Proceedings of 12th International Symposium on Methodologies for Intelligent Systems, ISMIS'00. Lectures Notes in Artificial Intelligence 1932. Springer-Verlag, Berlin (2000).
6. Sowa, J.F., Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole Publishing Co., Pacific Grove (CA), 1999.
7. Zarri, G.P., and Gilardoni, Structuring and Retrieval of the Complex Predicate Arments Proper to the NKRL Conceptual Language. In: Ras, Z, Michalewicz, M. (eds.): Foundations of Intelligent Systems - Proceedings of 9th International Symposium on Methodologies for Intelligent Systems, ISMIS'96. Lectures Notes in Artificial Intelligence 1079. Springer-Verlag, Berlin (1996).
8. Franconi, E., A Treatment of Plurals and Plural Quantifications Based on a Theory of Collections, Minds and Machines, 3 (1993) 453-474.
9. Lassila, O., Swick, R.R., eds., Resource Description Framework (RDF) Model and Syntax Specification. W3C, 1999 (http://www.w3.org/TR/REC-rdf-syntax/).
10. Brickley, D., Guha, R.V., eds., Resource Description Framework (RDF) Schema Specification. W3C, 1999 (http://www.w3.org/TR/WD-rdf-schema/).
11. Zarri, G.P., and Azzam, S., Building up and Making Use of Corporate Knowledge Repositories. In:), Plaza, E., Benjamins, R. (eds.): Knowledge Acquisition, Modeling and Management - Proceedings of the European Knowledge Acquisition Workshop, EKAW'97. Lectures Notes in Artificial Intelligence 1319. Springer-Verlag, Berlin (1997).

Using Agents for Concurrent Querying of Web-Like Databases via a Hyper-Set-Theoretic Approach

Vladimir Sazonov

Department of Computer Science
University of Liverpool, Liverpool L69 7ZF, U.K.
`http://www.csc.liv.ac.uk/~sazonov`
`V.Sazonov@csc.liv.ac.uk`
Tel: (+44) 0151 794 6792
Fax: (+44) 0151 794 3715

Abstract. The aim of this paper is to present a brief outline of a further step in the ongoing work concerning the hyper-set-theoretic approach to (unstructured) distributed Web-like databases and corresponding query language Δ. The novel idea in this paper consists in using dynamically created mobile agents (processes) for more efficient querying such databases by exploiting concurrently distributed computational resources, potentially over the whole Internet. A fragment of a calculus of querying agents based on Δ is presented.

1 Introduction

Querying and searching the World-Wide Web (WWW) or, more generally, *unstructured* or *Web-like Databases (WDB)* is one of the most important contemporary information processing tasks. There are several search engines such as Alta Vista, Lycos, etc., but their search can hardly be characterised as "goal-directed" and always "up to date". Also it is desirable not only to be able to find a list of Web-pages of potential interest, but to ask more complex queries allowing, additionally, reorganisation of Web data, as required. That is, the answer to a query should constitute a number of possibly newly created hyper-linked pages (re) constructed from some pages existing somewhere in WDB — very much in the same way as in a relational database. A new relation/WDB-page(s) (the answer to a query) is the result of reconstructing existing ones in the database. In fact, our approach to WDBs is *a natural generalisation of the traditional relational databases* and differs essentially from the other known considerations of semi-structured databases, being actually *(hyper) set-theoretic* one.

The starting point for this work was the characterisation of PTIME computability in terms of recursion over finite structures obtained by the author [32] and independently by N. Immerman, A. Livchak, M. Vardi and Y. Gurevich [18,20,26,42,18]. It should be mentioned, of course, the seminal previous work of R. Fagin [14] on describing NPTIME in terms of existential second-order logic.

Such results were also considered in a *framework of an abstract approach to query languages for relational databases.* The subsequent work of the author on Bounded Set Theory [33,34,36,23,22] is a natural continuation and generalisation of these results for the case of more flexible structures such as hereditarily-finite sets which are more suitable for providing mathematical foundations of *complex* or *nested databases.* Later these considerations absorbed, in [35], the idea of non-well-founded set (or *hyper-set*) theory introduced by P. Aczel [3]. This made the approach potentially applicable to (possibly distributed) *semistructured* or *Web-like databases* [24,25,37] with allowing cycles in hyper-links. Using *distributed, dynamically created agents* for more efficient querying of such databases by exploiting concurrently distributed computational resources (potentially over the whole Internet) is a further step to be developed within this framework.

Note, that our work *is not* intended to working out a *general theory* of agents. They are rather *a tool for achieving efficiency* in the querying process. However, an intermediate task consists of developing a new (or adapting some previously known) *theory or calculus of agents* suitable to our considerations. (Cf. references at the end of Section 4.) In this short paper we can only outline the main (essentially inseparable) ideas: *hyper-set approach to WDB* and to *querying WDB*, and *using agents* (in the context of this hyper-set framework). Also, in spite of any declared allusion to reality of WWW, which seems very useful and thought provoking, the present paper is highly abstract (and simultaneously informal and sketchy in some respects). But the author believes that the level of abstraction chosen is quite reasonable. The goal is not WWW itself, but some very general ideas around set-theoretic approach to WDB.

Our plan is as follows. Sect. 2 outlines the hyper-set approach to WDB. Sect. 3 is devoted to a brief but quite precise presentation of a hyper-set Δ-language and its extensions for querying WDB. Sect. 4 discusses informally using agents for concurrent and distributed querying WDB. Then a fragment of a calculus of agents for computing Δ-queries (a preliminary semiformal version) in Sect. 5 is presented. Finally, a Conclusion briefly summarises the whole paper.

2 An Outline of Hyper-Set Approach to WDB

The popular idea of a *semi- (or un-) structured data base* which, unlike a relational database, has no rigid schema, has arisen in the database community rather recently (cf. e.g. [1,2,7,27,8]), particularly in connection with the Internet. These databases can also be naturally described as *Web-like* databases (WDB) [24,25,37,38,39]. The best example of such a database is the World-Wide Web itself understood as a collection of Web-pages (`html`-files) distributed by various *sites* over the world and arbitrarily *hyper-linked.*

We adopt here a deliberately simplified, very abstract picture which, however, involves the *main feature* of WWW as based on hyper-links between URLs[1]

[1] Recall that in WWW URL = *Uniform Resource Locator*, i.e. *address* of a WWW-page (a specific html-file) on a Web-server.

of the Web-pages distributed potentially over the world. Having in mind set-theoretical approach to semistructured databases (arisen quite independently of WWW in [33] or, in somewhat different framework, [9,10]) and contemporary ideas on semistructured databases (cf. op. cit.), we consider that *the visible part* of a WDB-page, Page(u), with the URL u is a (multi)set $\{\!|l_1, l_2, \ldots, l_k|\!\}$ of words l_i (rather than a *sequence or list* $\langle l_1, l_2, \ldots, l_k\rangle$) where $u \xrightarrow{l_i} v_i$ represent all the outgoing hyper-links from u to v_i. The URLs v_i are considered as *non-visible* part of the *page* Page$(u) = \{l_1 : v_1, l_2 : v_2, \ldots, l_k : v_k\}$ (a set of pairs of the kind *label* : *URL* — an abstraction from an `html` file). Both l_i and corresponding v_i are present in this page having the URL u, but in the *main window* of a browser we do *not* see the URLs v_i; u being shown *outside* the window. Their role is very important for organisation of the information in WDB via *addressing/hyper-linking* but quite *different* from that of the words l_i, the latter carrying the *proper information.* All words l_i on a Web-page are considered as (names or labels of) hyper-links, possibly trivial ones (e.g. links to an empty or non-existing page). By analogy with WWW we could underline $\underline{l_i}$ if $u \xrightarrow{l_i} v_i$ and v_i is "non-empty", i.e. there exists at least one hyper-link $v_i \xrightarrow{m} w$ outgoing from v_i (and it therefore makes sense to "click" on $\underline{l_i}$ to see all these m).

Such Web-like databases are evidently represented as directed *graphs with labelled edges* (or *labelled transition systems*), the labels l serving as the names of hyper-links or *references* $u \xrightarrow{l} v$ between graph vertices (URLs) u and v. Equivalently, WDB may be defined a finite subset of $\mathsf{URL} \times \mathsf{Label} \times \mathsf{URL}$. As in any database, a query language and corresponding software querying system are needed to properly query and retrieve the required information from a WDB.

In principle, for intermediate theoretical goals, it makes sense to consider also simplified "label-free", "pure" framework with simple unlabelled hyper-links $u \to v$ and (corresponding oversimplified) WDB is just an arbitrary finite (directed) graph with Edges — URLs. Then WDB-page Page(u) with a URL u is just a set of URLs $\{u_1, \ldots, u_k\}$ such that $u \to v_i$ are *all* outgoing edges from u.

Let us assume additionally that each graph vertex (URL) u refers both to a *site* Site$(u) \in \mathsf{SITES}$ where the corresponding WDB-page is saved as an (`html`) file and to this WDB-page itself. Some URLs may have the same site: Site$(u_1) =$ Site(u_2). However it is not the most interesting case, it is quite possible when there exists only one site for the whole (in this case non-distributed) WDB.

In the *hyper-set-theoretic* approach adopted here each graph vertex u (URL [24,37], called also *object identity, Oid*, [1,2,7]) is considered as carrying some *information content* $(\!|u|\!)$ — the result of some abstraction from the concrete form of the graph. In principle, two Web-pages with different URLs u_1 and u_2 may have the same (*hereditarily*, under browsing starting from u_1 and u_2) visible content. This is written as $(\!|u_1|\!) = (\!|u_2|\!)$ or, equivalently, as $u_1 \sim u_2$ where $\sim$ is a *bisimulation equivalence relation* on graph vertices (which can be defined in graph-theoretic terms [3] independently of any hyper-set theory). Somewhat analogous considerations based on a bisimulation relation are developed in [7], but the **main idea of our approach** consists not only in respecting the bisimulation (or, actually, *informational equivalence*) relation, but in "considering" graph vertices *as abstract sets* $(\!|u|\!)$ within hyper-set theory [3]:

$$(\!|u|\!) = \{l : (\!|v|\!) \mid \text{WDB has a labelled hyper-link } u \xrightarrow{l} v\}. \qquad (1)$$

or in the simplified, "pure, label-free" semantics:

$$(\!|u|\!) = \{(\!|v|\!) \mid \text{WDB has a hyper-link } u \to v\}. \qquad (2)$$

Thus, $(\!|u|\!)$ is a (hyper-) set of all labelled elements $l : (\!|v|\!)$ (also hyper-sets, etc.) such that WDB has a hyper-link $u \xrightarrow{l} v$. This (actually uniquely satisfying (1)) set-theoretic *denotational semantics* of graph vertices has evidently a complete agreement with the meaning of the equality $(\!|u_1|\!) = (\!|u_2|\!)$ or bisimulation equivalence $u_1 \sim u_2$ briefly mentioned above. (Cf. details in [35,24,25,37].)

Unlike the ordinary (Zermelo-Frenkel) set theory, cycles in the membership relation are here allowed. Hence, a simplest such "cycling" set is $\Omega = \{\Omega\}$ consisting exactly of itself. Such sets must be allowed because hyper-links (in a WDB or in WWW) may in general comprise arbitrary cycles. We have *crucial theoretical benefits* from this hyper-set framework by using well understood languages (like Δ formally presented in Sect. 3) and ideas. Hyper-sets arising from graph vertices via (1) are quite natural and pose no essential conceptual difficulty.

Returning to the graph view, *querying* of a WDB may be represented as

– starting on a local site s (where the query $q(u)$ is evaluated) from an *initial* or *input URL u of a page* (saved as an `html` file on any remote site Site(u));
– *browsing* page-by-page via hyper-links (according to the query q);
– *searching* in pages arising in this process (according to the query q);
– *composing new (auxiliary — for temporary intermediate goals — or final) hyper-linked pages with their URLs* (on the basis of the previous steps) with one of these pages declared as *main answer* or *resulting (output or "title") page* for the query;
– all of this may result in *reorganising of the data* (at least locally on s by auxiliary pages located and hyper-linked between themselves and externally).

Of course, the user could habitually do all this job by hands. However, it is more reasonable to have corresponding implemented *query language and system* which would be able to formally express and evaluate any query q. This essentially differs from the ordinary search engines such as Alta Vista or Lycos, etc. Here we can use the advantages of our set theoretic view and of the corresponding language Δ [34,36,24,25], at least theoretically.

Again, the **key point of our approach** consists in co-existing and interplaying *two* natural and related *viewpoints* for such a query q: graph- and set-theoretic ones. The above described steps result in some transformation (essentially a local extension by new pages) of the original WDB (WWW) graph. Usually, in semistructured databases *arbitrary* graph transformations are allowed [1,2]. But if this process *respects the information content* $(\!|u|\!)$ of the input WDB vertices (URLs) u then it should be restricted to be *bisimulation invariant* (as it was done also independently in [7], but without a sufficient stress on set theory) and therefore it could be also considered as a *set theoretic operation* q, i.e.

$$q : (\!|\text{input URL}|\!) \longmapsto (\!|\text{output URL}|\!) = q(\!|\text{input URL}|\!).$$

Originally, this approach arose exactly as a set-theoretical view [33,34] with (at that time *acyclic*) graphs representing the ordinary or *well-founded hereditarily-finite sets* whose universe is denoted as **HF** [2] [6]. Any database state may be considered as a set of data each element of which is also a further set of data, etc. (with always *finite depth of nesting* which is equal 4 in the case of a "flat" *relational database*). A generalised universe of *anti-founded hereditarily-finite (hyper) sets* is called **HFA**, and any query is considered as a map (set-theoretic operation — a well understood concept) $q : \mathbf{HF} \to \mathbf{HF}$ or $q : \mathbf{HFA} \to \mathbf{HFA}$.

A (version of) purely set-theoretic *query language* Δ allowing expression of queries q to a WDB has been developed (with a natural and simplest syntax whose main feature is the use of *bounded* quantifiers $\forall x \in t$ and $\exists x \in t$ and other, essentially bounded, constructs; cf. e.g. [35,36,25,24] and Sect. 3 below for the details). This language has *two kinds of semantics* in terms of: (i) *set-theoretic operations* q over **HFA** (such us set union or "concatenation" of WDB-pages), etc. — *a high level* semantics *for humans*) and (ii) corresponding *graph (WDB) transformers* Q — (*a lower level* semantics *for program implementation*), with a commutative diagram (3) witnessing that both semantics agree

$$\begin{tikzcd} \mathbf{HFA} \arrow[r, "q"] & \mathbf{HFA} \\ \mathcal{WDB} \arrow[u, "(\!|-|\!)"] \arrow[r, "Q"] & \mathcal{WDB} \arrow[u, "(\!|-|\!)"'] \end{tikzcd} \qquad q(\!|\text{WDB}|\!) = (\!|Q(\text{WDB})|\!) \tag{3}$$

and Q is *bisimulation invariant* or *respecting informational equivalence*. Here $\mathcal{WDB}$ is the class of all WDBs (i.e. finite labelled or unlabelled graphs) with a distinguished URL (vertex) u in each, to which $(\!|-|\!)$ is actually applied.

The *expressive power* of this language and its versions was completely characterised in terms of PRIME- (both for **HF** and **HFA**) and (N/D)LOGSPACE- (for **HF**) computability over graphs via (3) [33,34,24,25,23,22]. It is because of flexibility, naturalness and reasonable level of abstractness of our set-theoretic approach to WDB (which seemingly nobody else used in the full power) such expressibility results where possible. Here it is probably suitable to mention, for the contrast, a quotation from [8], p. 14: "It should be noted that basic questions of expressive power for semistructured database query languages are still open".

[2] **HF** and **HFA** below may be considered in *two versions*: (i) *labelled* one when a set in the universe consists of labelled elements (which can be also such sets, or, may be, *urelements* or *atoms*), and (ii) *pure, atoms and labels-free* version — the ordinary pure sets of sets of sets, etc., the empty set being the simplest object. For simplicity, Δ-language and calculus of agents presented in Sect. 3 and 5, respectively, are devoted to the pure case. Labelled case would seemingly add nothing essential to our agents approach. Anyway, it is better to start with the simplest case.

A preliminary *experimental implementation* of a version of Δ [38] has been developed (when the author worked) in the Program Systems Institute of Russian Academy of Sciences (in Pereslavl-Zalessky) as a query language for the "real" WWW which is based on the steps of browsing, searching, etc. described above.

3 Hyper-Set $\boldsymbol{\Delta}$-Language for Querying WDB

3.1 Basic $\boldsymbol{\Delta}$-Language for the Pure HF or HFA[3]

Definition 3.1. Define inductively Δ^*-*formulas* and Δ^*-*terms* by the clauses

$$\langle \Delta^*\text{-terms}\rangle ::= \langle \text{variables}\rangle \mid \emptyset \mid \{a,b\} \mid \bigcup a \mid \{t(x) \mid x \in^{(*)} a \ \&\ \varphi(x)\}$$

$$\langle \Delta^*\text{-formulas}\rangle ::= a = b \mid a \in^{(*)} b \mid \varphi \ \&\ \psi \mid \neg\varphi \mid \forall x \in^{(*)} a \varphi(x)$$

where φ and ψ are any Δ^*-formulas, a, b and t are any Δ^*-terms and x is a variable not free in a. The parentheses around $*$ in $\in^{(*)}$ mean that there are two versions of the membership relation: $\in$ and its non-reflexive *transitive closure* $\in^*$. We write Δ when $\in^*$ is not used at all. The sub-language Δ corresponds to the *basic* [17] or *rudimentary* [21] set-theoretic operations.

From the database's point of view, (labelled version of) the language Δ is analogous to *relational calculus*, however it is devoted to *arbitrary sets of sets of sets*, etc. rather than to relations — also *sets of a specific kind.*

The main idea is that Δ-formulas involve only *bounded* quantification $\forall x \in^{(*)} a$ and $\exists x \in^{(*)} a \rightleftharpoons \neg\forall x \in^{(*)} a\neg$ with the evident meaning. All other constructs of Δ are also in a sense "bounded". But, for example, the unrestricted *power-set operation* $\mathcal{P}(x) \rightleftharpoons \{y : y \subseteq x\}$ is considered as intuitively "unbounded", "impredicative", computationally intractable, and therefore it is not intended to be taken as *natural* extension of Δ.

For simplicity, as common, we shall use set variables, Δ^*-terms and formulas *both as syntactic objects and as denotations of their values* in **HF** (**HFA**). For example, Δ-separation $\{x \in a : \varphi(x)\}$ for $\varphi \in \Delta$ gives 'the set of all x in the set a for which $\varphi(x)$ holds' and is a special case of the construct $\{t(x) : x \in a \ \&\ \varphi(x)\}$ = 'the set of all values of $t(x)$ such that ...'. Also $x \in \{a, b\}$ iff $x = a$ or $x = b$, $x \in \bigcup a$ iff $\exists z \in a(x \in z)$ and $\in^*$ is a transitive closure of the membership relation $\in$ on **HF** (**HFA**), i.e. $x \in^* y$ iff $x \in x_1 \in x_2 \in \ldots \in x_n \in y$ for some $n \geq 0$ and $x_1, \ldots, x_n$ in **HF** (**HFA**). In particular, define in Δ^* the *transitive closure of a set* y as $\mathrm{TC}(y) \rightleftharpoons \{y\} \cup \{x : x \in^* y\}$.

Any Δ-term $t(\bar{x})$ evidently defines a set-theoretic operation $\lambda\bar{x}.t(\bar{x})$: $\mathbf{HFA}^n \to \mathbf{HFA}$. For example, let $u \cup v \rightleftharpoons \bigcup\{u, v\}$, $u \cap v \rightleftharpoons \{x \in u : x \in v\}$, $u \setminus v \rightleftharpoons \{x \in u : x \notin v\}$, $\{u\} \rightleftharpoons \{u, u\}$, $\{u_1, u_2, \ldots, u_k\} \rightleftharpoons \{u_1\} \cup \{u_2\} \cup \ldots \cup \{u_k\}$. Ordered pairs and related concepts are defined in Δ as $\langle u, v\rangle \rightleftharpoons \{\{u\}, \{u, v\}\}$, or (in the labelled case), more naturally from a practical point of view, as $\{\mathrm{Fst} : u, \mathrm{Snd} : v\}$ for some two special labels (attributes) Fst and Snd, etc.

[3] See [34] for generalisation of $\boldsymbol{\Delta}$ for the case of **HF** and **HFA** with urelements (atoms) and labels (attributes).

3.2 Extensions of $\varDelta^*$

There are important extensions of $\varDelta^*$-language corresponding to PTIME or (N/D)LOGSPACE computability over **HFA**, resp., **HF**, as we mentioned in Sect. 2. Due to lack of space, we will consider only those giving rise to PTIME.

Decoration (Aczel, [3])

$$\mathbf{D} : \mathbf{HFA} \to \mathbf{HFA}, \quad \mathbf{D}(\langle g, v\rangle) = (\!| v |\!)_g$$

with $g \in$ **HFA** any graph (set of ordered pairs in set-theoretic sense). The decoration operation $\mathbf{D}$ embodies Aczel's **Anti-Foundation Axiom**.

Collapsing (Mostowski, [6])

$$\mathbf{C} : \mathbf{HF} \to \mathbf{HF}$$

is partial case of $\mathbf{D} : \mathbf{HFA} \to \mathbf{HFA}$ when g is any *well-founded*, i.e. *acyclic* graph (in the finite case considered).

Recursive $\varDelta$-Separation term construct **Rec**:

$$\textbf{the-least}\ p.[p = \{x \in a : \varphi(x, p)\}]$$

for $\varphi \in \varDelta$ *positive* (and with an additional requirement) on set variable p.

4 Using Agents for Concurrent Querying WDB

What is new in the present approach. The *main problem* with implementing the set-theoretic language $\varDelta$ is that (potentially) a *significant amount of browsing* may be required during query evaluation. In the real Internet the result of each mouse click on a hyper-link (downloading a page from remote site) is delayed by several seconds at best and the whole time of querying may take hours (despite all other pure computation steps being much faster). Therefore, some *radical innovation in the implementation is necessary* to overcome the problem of multiple browsing via the Internet (in a *non-efficient, sequential* manner).

A well-known technique, used by most search engines such as Alta Vista, is that of creating (and permanently renewing) an *index file* of the WWW to which queries are addressed and some parts of which may be actually rather old.

The approach described here is aimed at "goal-directed" querying the *real* Web, *as it is* at the current moment, where *reorganising the data* (not only searching) is an additional and crucial feature of the query language $\varDelta$.

As a reasonable solution, we suggest using **Dynamic Agent Creation**. The set-theoretic language $\varDelta$ appears very appropriate for this goal.

Agents as $\varDelta$-terms. Each agent, i.e. essentially a $\varDelta$-term (= a query $q = q(u)$ written in $\varDelta$-language) having an additional feature of an *active behaviour*, when starting querying from "his" local site s, may also *send* (via the Internet) to remote sites $s_1, s_2, \ldots$ some appropriate sub-terms $p_1, p_2, \ldots$ which, as *agents*, will work analogously on these remote sites, i.e., if necessary, will send new *descendant agents*, etc. Eventually, the main agent will collect all the data obtained from this process. Potentially, the *whole Internet* would *concurrently participate* in the *distributed* evaluation of the given query. We expect that this will essentially *accelerate querying*. One of medium-term goals is to establish the truth of this expectation in a theoretical framework. Another goal is producing an experimental agent based implementation of the language Δ to check practically this expectation and to get more experience for further developing this approach.

A Typical Example. To calculate the set-theoretic Δ-term (in unlabelled case)

$$q = \{p(v) \mid v \in a\},$$

i.e. the "set of all $p(v)$ such that v is in a", this term, as an "agent", *sends many descendant agents/sub-terms* $p(v)$ for all (URLs) v contained on the page a to the various, probably *remote*, sites Site(v) of the WWW to which all these (URLs) v refer. When each agent $p(v)$ finishes "his" computation (possibly with the help of their descendant agents), "he" will send the result to the "main agent" q, "who" will appropriately collect all the results together as a new (URL — the value of q — and corresponding `html` file of a) Web-page containing all the computed URLs $p(v)$ for all $v \in a$. Each agent q, $p(v)$, for all $v \in a$, etc. will resolve "his" own task, which may be probably not so difficult and burdensome for the site resources where the agent works, since "his" descendant agents will do the rest of the job possibly on other sites. If, by some reason, the "permission is denied" to send a descendant agent to a remote site, the parent agent will also take corresponding part of the job and will do it "himself" from "his" local site — by browsing, searching, etc.

The further and most crucial goal of the described work therefore consists in *formalising the ideas above*. It may be done in the form of a specially elaborated (or suitably adapted for the query language Δ considered here) *calculus of agents* which describes as descendant agents are creating, moving around, doing their job, communicating their result to the "senior" agents by which they were created, until the result of the initial query is obtained. In particular, this will give *agent based graph transformer operational semantics* for the language Δ. Some appropriate version of the corresponding agent calculus (for a partial case covering the above example) will be presented in Sect. 5. We realize that this is probably not the unique possible version. Cf. discussion below.

Correctness of this semantics with respect to natural (hyper) set-theoretical semantics must be proved. It should be shown rigorously that the *time complexity* of this approach with agents is really better than that without agents (i.e. essentially with only one main agent which creates no sub-agents and makes all "his" job of browsing, searching and creating pages alone). In particular, it is necessary to formulate and develop a reasonable approach to time and space complexity for agent based querying (cf. the next paragraph) and then to estimate and compare time and space complexity of queries either when agents are used or not. In the ideal, the resulting agent calculus should be *capable of being implemented* as a working *query system* to a WDB, say, to the WWW.

Note that an appropriate abstraction from querying of the WWW should be found. For example, in reality, each step of browsing requires some physical, actually unpredictable time. For simplicity it may be taken that each step of browsing or exchanging information between agents takes *one unit of time* and all other computation steps (of searching some information trough Web-pages and composing new ones) cost nothing (in comparison with the browsing). Also when several agents are sent to the same Web-site for their work their activity may be considered either as sequential or in an interleaving manner according to communication with other agents. Space complexity also may be considered in a simplified version, e.g. as the maximum number of agents working simultaneously. The simultaneity concept should be also defined (approximated) appropriately as the run of time in the model may not correspond to the real one, as we noted above. On the other hand, how much of memory resource each agent needs on its site (server) could additionally be taken into account. All of this seems has no canonical or simple, most elegant theoretical solution.

Some related work (i) on *communication and concurrency* by R. Milner and P. Aczel, [28,4,5], (ii) on *mobile agents (ambients)* by R. Milner et al. [29], L. Cardelli [8], C. Fournet et al. [16], (iii) on *logical specification* of agents' behaviour by M. Fisher and M. Wooldridge [15], (iv) on *distributive querying* by D. Suciu [40,41] and A. Sahuguet et al. [31], (v) on corresponding application to agents of Y. Gurevich's *Abstract State Machines* [13], (vi) on *Mobile Maude* by J. Meseguer et al. [12], (vii) on *Nomadic Pict* [30], (viii) on a project *LOGIC Programming for the World-Wide WEB* by A. Davison et al. [11] and probably many others could be useful in our specific set-theoretic context either ideologically, or by direct using corresponding formalisms or their analogues.

However, in our very concrete situation it seems reasonable to present a calculus of agents from the principles which the query language Δ almost immediately "dictates" according to its natural and clear semantics, postponing for a while comparison with other formalisms and ideas, because here we will present only rather preliminary, even not completely rigorous version of a calculus.

Further perspectives from the practical point of view. It is clear, that *the whole approach depends on giving permissions* by various Web sites for agents to work there. For a WDB on a local net or on an *Intra*net this problem can be resolved by an agreement with the administrator of the net. However, for the case of the global Internet the whole Internet community should agree on a *general standard* for such permissions with an appropriate guarantee of no danger from the "arriving" agents. Of course this may depend on whether this framework will be sufficiently successful for local nets, as well as on the current state of affairs concerning programming of distributed computation in the Internet.

5 A Fragment of a Calculus of Agents for Computing Δ-Queries (Preliminary Semiformal Version)

According to our informal description in the previous section, each Web-site s (server, or even any computer with a Web-browser)[4] may contain agents/queries $q??$, or $q = ??$ working on the site s (where q, with some exceptions, is a Δ-term or Δ-formula; the general form of an agent/query is presented in (4)). All queries on a site s constitute a *queue* $\tilde{q}$ according to which (from left to right) they should be evaluated. We consider the expression $s\,\tilde{q}$ as a current *state* of a site s w.r.t. (the state of) its queries in $\tilde{q}$.

The *double question mark* "??" means that the query/agent is waiting for the server s to start (or allow) executing it, while a *single question mark* "?" corresponds to an agent q which has already started to work, but at present is *awaiting results* from its subqueries (descendant agents) which have been sent to other sites. Also, we use single "?" for so-called "passive" queries which are only waiting for results from "active queries" (cf. below).

Thus, the sequence $\tilde{q}$ consists of *active agents/queries* (syntactically more complicated than just "$q = ??$") formally written in round brackets as:

$$(q_i = \epsilon_i \to s_i\, id_i), \quad i = 1, \ldots, m. \tag{4}$$

Here "$\to s_i\, id_i$" is read as "send the result of the query to the site s_i by the address id_i". Expressions like *id* of our agents metalanguage are *query identities* of various "passive" subqueries (to be considered below — they are parts of ϵ-expressions of other, actually parent agents/queries also of the form (4)). These identities are needed to distinguish *occurrences* of "passive" subqueries and also to serve as their *addresses* in the corresponding queues (the current implicit queue in the site s_i, in the case of (4)). All query identities constitute a set Id of finite strings consisting of some standard computer keyboard symbols starting with, say, # and containing no more #s.

Some special identities written as #`browser-of-` < `user` > are reserved as *special, exceptional cases*, used for queries of the form

$$(q_i = ?? \to s'\#\texttt{browser-of-} < \texttt{user} >)$$

in the site-queue $s\,\tilde{q}$ initiated by some *(human) user/agent* working on another site s' (his PC). This query will be started for execution on the site s (according to our formal calculus of agents to be partially described below). Let us assume that some URL u_i will be obtained by evaluating (probably in many steps) the query "$q_i = ??$", giving rise to the result "$q_i = u_i$". Then, the remaining part "$\to s'$#`browser-of-` < `user` >" means that the resulting Page(u_i) should be opened in a new window of a browser on the site s' with some additional special subwindow in the browser where the answer "`QUERY RESULT OF <user>`

[4] Everything is still considered in a very abstract way; our Web or Internet terminology is used mainly to provide a general intuition rather than a complete or very precise correspondence to standard concepts.

`IS:` $q_i = u_i$" to the query is explicitly written for the user's convenience to understand why this (perhaps suddenly appearing because of a possible time delay) browser window and its content Page(u_i) arose.

Let us *define in general* what the expression ϵ_i in (4) may be:

- double question mark "??", or
- just some *resulting* $u_i \in \mathsf{URL}$, if q_i is Δ-term (a normal query), or a *resulting* boolean value true or false, if q_i is Δ-formula (a boolean query), or
- starting with a single question mark "?" and possibly with a URL, a *sequence* of auxiliary *passive queries* or corresponding *answers* in square brackets

$$[id_r : r = ?] \quad \text{or} \quad [id_r : r = u_r], \quad \text{or} \quad [id_r : r = U_r], \tag{5}$$

  with $u_r \in \mathsf{URL}$, U_r a finite set of URLs or some simple expressions consisting of URLs (in the last two cases, the "id :" will usually be omitted); these passive queries or answers are needed to calculate the answer to the main query (4) which is of the form

$$(q_i = ?[\ldots]\ldots[\ldots] \to s_i\, id_i) \quad \text{or} \quad (q_i = ?url[\ldots]\ldots[\ldots] \to s_i\, id_i); \tag{6}$$

  moreover, all query identities id_r should be *different* in passive queries of the whole queue $\tilde{q}$ (to serve as unique addresses of these passive queries for sending the answers from other sites).

Here a single "?" at the beginning of ϵ_i indicates that the result of q_i in (4) or (6) is still unknown. Even if we have some url in (6), the content of the Page(url) may be not completed yet. To get the result of q_i in (6) some further actions of this agent may be required even if all passive subqueries are already answered and replaced by the results of the form $[r = u_r]$ or $[r = U_r]$.

As the server s may be "busy" with its own work or evaluating queries from the queue $\tilde{q}$, it is indeterminate when it will execute the first "waiting" query q_i?? or q_i? (with queries for all $j < i$ having already some final answers $u_j \in \mathsf{URL}$ or a boolean value true or false, for boolean queries, or waiting the results from their descendant subagents which already have been sent to some sites by the agent q_j with the help of the server s). Therefore we should have nondeterministic, asynchronous and concurrently applied *reduction rules* for the extended sites $s\tilde{q}$.

Note that all Δ-queries q_i should be closed terms or formulas with free variables substituted by some urls (the addresses of WDB-pages which simultaneously denote corresponding sets $(\!|url|\!)$ according to (1) or (2) — in our present "pure" case). However, q_i may be also a specially created agent which is not formally a Δ-term.

Some q_i may be just a constant $u \in \mathsf{URL}$. This query does not require evaluation. As a URL, it is the final result of evaluation of the query. Therefore let us introduce the **trivial rule** for such *atomic* q_i:

$$s\tilde{q}(u = ?? \to s'id')\tilde{q}' \longmapsto s\tilde{q}(u = u \to s'id')\tilde{q}'. \tag{7}$$

The following **double structural rule** (*inseparable* into two independent parts; same for other "multiple rules" below presented in curly brackets), for the sequential replacement of queues in two sites, shows what happens after a query agent q_i has received an answer u_i (say, as in the trivial case of (7)):

$$\left\{\begin{array}{ccc} s\tilde{q}(q_i = u_i \to s_i id_i)\tilde{q}' & \longmapsto & s\tilde{q}\tilde{q}', \\ s_i\tilde{q}'' & \longmapsto & s_i\langle u_i \to id_i\rangle\tilde{q}'' \end{array}\right\} \tag{8}$$

(if $s = s_i$ then it is naturally assumed that $\tilde{q}'' = \tilde{q}\tilde{q}'$ should hold) where the *substitution operator* $\langle u_i \to id_i\rangle$ means that the (unique) subquery $[id_i : \ldots = ?]$ (of some parent agent) with the same id_i should be found in an ϵ part of $(q = \epsilon \to \ldots)$ occurring in $\tilde{q}''$ and corresponding "?" should be replaced by u_i with id_i possibly omitted as already used and unnecessary anymore, giving result $[\ldots = u_i]$ in ϵ in the place of the passive query $[id_i : \ldots = ?]$ occurrence.

5.1 Rules for the Query "$\{p(v) \mid v \in a\} = ??$"

The following three rules are devoted to the query "$\{p(v) \mid v \in a\}??$", informally considered in the previous section.

The first rule postpones the main query in favour of $(a = ?? \to s\,id_a)$ sent to the same site s, last in the queue:

$$\begin{array}{l} s\tilde{q}(\{p(v) \mid v \in a\} = ?? \to \ldots)\tilde{q}' \longmapsto \\ \qquad s\tilde{q}(\{p(v) \mid v \in a\} = ?[id_a : a = ?] \to \ldots)\tilde{q}'(a = ?? \to s\,id_a) \end{array}$$

As discussed above, the part "$\to s\,id_a$" shows where the result should be sent; here, to the same site and queue to the passive query (in the main query) labelled by id_a. Note, that expressions $[id_a : a = ?]$ and $(a = ?? \to s\,id_a)$ play different roles in the queue (passive and active, respectively). When the latter will be resolved, with some result $u_a \in \mathsf{URL}$, it can be annihilated, as in (8), after the "?" in the former expression will be replaced by the result u_a obtained.

The second, multiple rule has additionally a *premise condition*[5] assuming that on the site s the query/agent $(\{p(v) \mid v \in a\} = ?[a = u_a] \to \ldots)$ (already "knowing" that u_a is the value of a) first inspects the content of the Page(u_a) (say, with the help of a browser on the site s). Then, depending on the result in the premise, the following multiple rules under the horizontal line are performed:

$$\frac{\mathrm{Page}(u_a) = \{u_1, \ldots, u_k\}}{\left\{\begin{array}{l} s\tilde{q}(\{p(v) \mid v \in a\} = ?[a = u_a] \to \ldots)\tilde{q}' \longmapsto \\ s\tilde{q}(\{p(v) \mid v \in a\} = ?[a = u_a][id_1 : p(u_1) = ?] \ldots [id_k : p(u_k) = ?] \to \ldots)\tilde{q}' \\ \qquad \mathrm{Site}(u_i)\tilde{q}_i \longmapsto \mathrm{Site}(u_i)\tilde{q}_i(p(u_i) = ?? \to s\,id_i), \quad i = 1, \ldots, k. \end{array}\right\}}$$

Again, the main query is postponed until new descendant agents of the form $(p(u_i) = ?? \to s\,id_i)$ (depending on the result u_a of the query "$a = ??$", or, more precisely, on the content of Page(u_a)) are created and sent to the sites Site(u_i) (with their current queues $\tilde{q}_i$) to work there and to calculate values of $p(u_i)$.

[5] We could use instead an if-then construct or just composition of some actions.

Finally, the following rule allows the complete evaluation of the query "$\{p(v) \mid v \in a\} = ??$":

$$\frac{\mathrm{Page}(u_a) = \{u_1, \dots, u_k\}, \quad \texttt{new}\ \mathrm{Page}(u) = \{u'_1, \dots, u'_k\}}{\begin{array}{c} s\tilde{q}(\{p(v) \mid v \in a\} = ?[a = u_a][p(u_1) = u'_1] \dots [p(u_k) = u'_k] \to \dots)\tilde{q}' \longmapsto \\ s\tilde{q}(\{p(v) \mid v \in a\} = u \to \dots)\tilde{q}' \end{array}}$$

where u is a *new* URL on the site s ($\mathrm{Site}(u) = s$) such that $\mathrm{Page}(u) = \{u'_1, \dots, u'_k\}$. It is assumed that the current agent, with all passive queries answered,

$$(\{p(v) \mid v \in a\} = ?[a = u_a][p(u_1) = u'_1] \dots [p(u_k) = u'_k] \to \dots),$$

will create this new page on its site s which will automatically receive a URL u (depending on the directory where corresponding `html` file with this page will be saved and, of course, on the name of this file — as usual for URLs).

5.2 Rules for the Transitive Closure TC

The rules for calculating $\mathrm{TC}(a)$ are based on the following simple ideas. Starting from the calculation of URL $u_a = a$, numerous identical copies of a special agent TC' are sent step-by-step (first by TC, and then by descendant TC's) to (the sites of) descendants v of the vertex u_a in the given graph of the WDB. Each copy $\mathrm{TC}'(v)$ gathers URLs immediately accessible from v and sends them, with the information how they were obtained, to the main agent $\mathrm{TC}(a)$ which collects all of this information in a newly created file $\mathrm{Page}(u)$ with a URL u. When all URLs reachable from u_a are collected in $\mathrm{Page}(u)$, this parallel process is considered finished, and gives the result u.

The first rule is as in the previous subsection:

$$\begin{array}{l} s\tilde{q}(\mathrm{TC}(a) = ?? \to \dots)\tilde{q}' \longmapsto \\ \qquad s\tilde{q}(\mathrm{TC}(a) = ?[id_a : a = ?] \to \dots)\tilde{q}'(a = ?? \to s\, id_a) \end{array}$$

The second, multiple rule has an additional *premise condition*:

$$\frac{\mathrm{Page}(u_a) = \{u_1, \dots, u_k\} \quad \texttt{new}\ \mathrm{Page}(u) = \{u_a \to u_1, \dots, u_a \to u_k\}}{\left\{\begin{array}{l} s\tilde{q}(\mathrm{TC}(a) = ?[a = u_a] \to \dots)\tilde{q}' \longmapsto \\ \qquad s\tilde{q}(\mathrm{TC}(a) = ?u[a = u_a][id_{\mathrm{TC}} : \mathrm{TC}'(\ ) = ?] \to \dots)\tilde{q}' \\ \mathrm{Site}(u_i)\tilde{q}_i \longmapsto \mathrm{Site}(u_i)\tilde{q}_i(\mathrm{TC}'(u_i) = ?? \to s\, id_{\mathrm{TC}}), \quad i = 1, \dots, k. \end{array}\right\}}$$

Again, the main query is postponed until new descendant agents of the form $(\mathrm{TC}'(u_i) = ?? \to s\, id_i)$ (depending on the result u_a of the query "$a = ??$") are sent to the sites $\mathrm{Site}(u_i)$ (with their current queues $\tilde{q}_i$) to work there and to calculate values of $\mathrm{TC}'(u_i)$. Here $\mathrm{TC}'(\ )$ is a new agent generated by TC. URLs u_i serve as *parameters* for TC'. We also use $\mathrm{TC}'(\ )$ *without parameters* in the corresponding passive queries. Further, u is a *new* URL on the site s ($\mathrm{Site}(u) = s$) such that (at the beginning of calculating of $\mathrm{TC}(a)$) $\mathrm{Page}(u) = \{u_a \to \mathrm{Page}(u_a)\} = \{u_a \to u_1, \dots, u_a \to u_k\}$ (a set of arrows between URLs).

We see that the format of ϵ expressions is slightly extended in comparison with the previous subsection. However here we seemingly have an answer u to the question "TC(a) = ??", the question mark before u indicates that *this answer is only a partial one.* More precisely, it is partial only in the content of the corresponding Page(u) ($= \{u_a \to u_1, \ldots, u_a \to u_k\}$ at the first moment). It should be extended with the help of descendant agents of the form TC$'(v)$.

Moreover, we need to consider, *more generally*, that the answer to any subquery $[\mathrm{TC}'(v) = ?]$ (first, $v = u_i$ from the above rule) is the set of arrows between URLs $U = \{v \to \mathrm{Page}(v)\} = \{v \to v_1, \ldots, v \to v_n\}$, where $\mathrm{Page}(v) = \{v_1, \ldots, v_n\}$ (we let $U = \{v \to \}$ if $\mathrm{Page}(v) = \emptyset$, i.e. v has no children) whereas in the previous subsection it was assumed that only some URLs may be query answers.

The third, also multiple rule describes the behaviour of the agent TC$'$ which (i) creates a set of arrows $v \to v_i$ outgoing from the URL v under consideration (as described above), and (ii) propagates its copy to sites Site(v_i). From each site where (a copy of) TC$'$ is working, it sends the results to the same place $s\,id_{\mathrm{TC}}$.

$$\frac{\mathrm{Page}(v) = \{v_1, \ldots, v_n\} \quad U = \{v \to \{v_1, \ldots, v_n\}\}}{\left\{\begin{array}{l} \mathrm{Site}(v)\,\tilde{q}(\mathrm{TC}'(v) = ?? \to s\,id_{\mathrm{TC}})\tilde{q}' \longmapsto \\ \qquad\qquad\qquad \mathrm{Site}(v)\,\tilde{q}(\mathrm{TC}'(v) = U \to s\,id_{\mathrm{TC}})\tilde{q}' \\ \\ \mathrm{Site}(v_i)\tilde{q}_i \longmapsto \mathrm{Site}(v_i)\tilde{q}_i(\mathrm{TC}'(v_i) = ?? \to s\,id_{\mathrm{TC}}), \quad i = 1, \ldots, n. \end{array}\right\}}$$

The rule continuing partial evaluation of the query "TC(a) = ??" (extending the content of Page(u)):

$$\frac{\mathrm{Page}(u) := \mathrm{Page}(u) \cup U}{\begin{array}{c} s\,\tilde{q}(\mathrm{TC}(a) = ?u[a = u_a][id_{\mathrm{TC}} : \mathrm{TC}'(\,) = U] \to \ldots)\tilde{q}' \longmapsto \\ s\tilde{q}(\mathrm{TC}(a) = ?u[a = u_a][id_{\mathrm{TC}} : \mathrm{TC}'(\,) = ?] \to \ldots)\tilde{q}' \end{array}}\,.$$

Here u is (intended to be) the URL created above on the site s ($\mathrm{Site}(u) = s$) such that Page(u) is the union of the old version and the set of arrows, say, $U = U_i = \{u_i \to \mathrm{Page}(u_i)\}$. The passive query $[id_{\mathrm{TC}} : \mathrm{TC}'(\,) = ?]$ appears again to wait some new U from other TC$'(v)$.

Finally, we present a **rule to complete the calculation of** TC(a):

$$\frac{\left\{\begin{array}{c} \text{Assuming all participating URLs in Page}(u)\text{ occur as the left ends of arrows,} \\ \textbf{the optional action:}\text{ omit in this page all arrows} \\ \text{and duplication of all remaining URLs} \end{array}\right\}}{s\,\tilde{q}(\mathrm{TC}(a) = ?u[a = u_a][id_{\mathrm{TC}} : \mathrm{TC}'(\,) = ?] \to \ldots)\tilde{q}' \longmapsto s\,\tilde{q}(\mathrm{TC}(a) = u \to \ldots)\tilde{q}'}\,.$$

The premise condition guarantees that Page(u) is completed as required, giving the whole transitive closure of a (or u_a), possibly with some arrows making Page(u) a graph. In the latter case, given any a and b, we could use the corresponding versions of TC(a) and TC(b) to calculate the bisimulation relation between corresponding graphs and then, if needed, to evaluate also queries "$(a = b)$??" and "$(a \in b)$??".

We postpone considering rules, their correctness and conditions and possible exceptions of applicability for the above and other constructs of the Δ-language until a more advanced version of this paper will be prepared (with a bigger volume allowed). Even in this form they demonstrate how the mobile and propagating agents can work over a (potentially World-Wide) distributed Web-like hyper-set database in a *concurrent, asynchronous and distributed manner* to evaluate set-theoretically described queries. After developing a full version of the calculus of "Δ-agents" both *theoretical* and *practical* (directed towards implementation) investigations can be carried out independently.

5.3 What Makes Agents Active, or How Could They Be Implemented?

Of course, agents, as they are described above, are only syntactic expressions. To make them active and really mobile, concurrent and working distributively, we need a unique program *Distributed Query Evaluator* DQE installed (desirably) *on each* WDB (or WWW) site which will fulfil all the necessary rules of manipulating queues (and agents in them). Of course, it should be sufficiently flexible to work even if some sites do not have installed DQE or do not currently allow our agents. Also it should be able to redirect some agents to other neighbour or "friendly" sites if the queue on the current site is too long, etc. Only in this case will the whole enterprise of making querying in Δ *considerably more efficient* according to the ideas presented in this paper be successful. It is really a difficult problem to make such a program which the whole Internet community and computer and software companies will adopt like well-known browsers (Internet Explorer or Netscape). Anyway, any new step in such a direction (theoretical, experimental, practical) **to make the Internet a big community of computers helping one another** seems a reasonable (much more general than our present) task.

6 Conclusion

The starting point for this research was a mostly theoretical one related to descriptive complexity theory and to extending its methods to relational, nested, complex and Web-like distributed databases grounded on (hyper) set theory. Our present work is also theoretical, but it is directed towards the problem of developing a necessary radical step for more efficient implementation. Considering dynamically created agents *in full generality*, working and communicating concurrently, asynchronously and distributively over the Internet *is not* the immediate goal of research here. We rather represent a *machinery for achieving the efficiency of querying in the present (hyper)set approach.* As a result, this work may be of direct benefit to research communities on semi-structured and distributed databases and on multi-agent systems. It also has the potential of longer term benefit to the Internet community. More concretely, it may lead to better understanding of the basic principles of efficient querying of WDB and to an experimental prototype of such a query system.

Acknowledgements. The author is very grateful to Michael Fisher, Alexander Bolotov, Chris Brand and Grant Malcolm for fruitful discussions on the approach presented and for the help in polishing English of this paper. He is especially thankful to his colleagues from Pereslavl-Zalessky (Russia) Alexander Leontjev, Alexei Lisitsa and Yuri Serdyuk actively participating in set-theoretic approach to WDB.

References

1. S. Abiteboul, Querying semi-structured data. *Database Theory — ICDT'97, 6th International Conference*, Springer, 1997, 1–18
2. S. Abiteboul and V. Vianu, Queries and computation on the Web. *Database Theory — ICDT'97, 6th International Conference*, Springer, 1997
3. P. Aczel, *Non-Well-Founded Sets*, CSLI Lecture Notes. No **14** (1988)
4. P. Aczel, Final Universes of Processes, *9th Internat. Conf. on Math. Foundations of Programming Semantics*, ed. S.Brookes, et al. Springer LNCS **802** (1994), 1-28
5. P. Aczel, Lectures on Semantics: The initial algebra and final coalgebra perspectives, *Logic of Computation*, ed. H. Schwichtenberg, Springer (1997)
6. J.K. Barwise, *Admissible Sets and Structures*. Springer, Berlin, 1975
7. P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, A query Language and Optimisation Techniques for Unstructured Data. *Proc. of SIGMOD*, San Diego, 1996
8. L. Cardelli, Semistructured Computation, Febr. 1, 2000; and other papers on mobile ambient calculi available via `http://www.luca.demon.co.uk/`
9. E. Dahlhaus and J.A. Makowsky, The choice of programming primitives for SETL-like programming languages, Proceedings of ESOP'86 (European Symp. on Programming, Saarbrüken, March 17-19, 1986), B. Robinet and R. Wilhelm eds., LNCS 213 (1986), 160–172.
10. E. Dahlhaus and J.A. Makowsky, Query languages for hierarchic databases, Information and Computation vol 101.1, (1992), pp. 1–32.
11. A. Davison, L. Sterling and S.W. Loke, *LogicWeb: LOGIC Programming for the World-Wide WEB*, `http://www.cs.mu.oz.au/~swloke/logicweb.html`
12. F. Durán, S. Eker, P. Lincoln, and J. Meseguer, Principles of Mobile Maude, LNCS, **1882**, , Springer-Verlag. 2000, 73–85.
13. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz, On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines, Springer LNCS **1912** 2000, 131–151,
14. R. Fagin, Generalized first order spectra and polynomial time recognizable sets, *Complexity of Computations*, SIAM — AMS Proc. **7**, 1974, 43–73
15. M. Fisher and M. Wooldridge, On the Formal Specification and Verification of Multi-Agent Systems. *Intern. J. of Cooperative Information Systems*, 6(1), 1997
16. C. Fournet, G. Gonthier, J-J. Lévy, L. Maranget, and D. Rémy, A Calculus of Mobile Agents, *CONCUR'96*, Springer LNCS **1119** (1996), 406–421
17. R.O. Gandy, Set theoretic functions for elementary syntax. Proc. Symp. in Pure Math. Vol. **13**, Part II (1974), 103–126.
18. Y. Gurevich, Algebras of feasible functions. *FOCS* **24** (1983), 210–214
19. Y. Gurevich at al., Web-site on ASM: `http://www.eecs.umich.edu/gasm/`
20. N. Immerman, Relational queries computable in polynomial time, *Proc. 14th. ACM Symp. on Theory of Computing*, (1982) 147–152

21. R.B. Jensen, The fine structure of the constructible hierarchy. Ann. Math. Logic **4** (1972), 229–308.
22. A. Leontjev and V. Sazonov, Capturing LOGSPACE over Hereditarily-Finite Sets, *FoIKS'2000*, Springer LNCS **1762** (2000), 156–175
23. A. Lisitsa and V. Sazonov, Delta-languages for sets and LOGSPACE computable graph transformers. *Theoretical Computer Science* **175**, 1 (1997), 183–222
24. A. Lisitsa, and V. Sazonov, Bounded Hyper-set Theory and Web-like Data Bases, *KGC'97*, Springer LNCS **1289** (1997), 172–185
25. A. Lisitsa and V. Sazonov, Linear ordering on graphs, anti-founded sets and polynomial time computability, *Theoretical Computer Science* **224**, 1–2 (1999) 173-213
26. A.B. Livchak, Languages of polynomial queries. *Raschet i optimizacija teplotehnicheskih ob'ektov s pomosh'ju EVM*, Sverdlovsk, 1982, p. 41 (in Russian)
27. A.O. Mendelzon, G.A. Mihaila, T. Milo, Querying the World Wide Web, Int. J. on Digital Libraries 1(1): 54–67 (1997)
28. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989
29. R. Milner, RJ. Parrow and D. Walker, A calculus of mobile processes, Parts 1–2, *Information and Computation*, 100(1), 1–66, 1992 (and other papers on Action calculi and the pi-calculus available via `http://www.cl.cam.ac.uk/users/rm135/`)
30. Nomadic Pict, `http://www.cl.cam.ac.uk/users/pes20/nomadicpict.html`
31. A. Sahuguet, B. Pierce and Val Tannen, Distributed Query Optimization: Can Mobile Agents Help? 2000, cf. `http://db.cis.upenn.edu/Publications/`
32. V. Sazonov, Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, **16** (7) (1980) 319–323
33. V. Sazonov, Bounded set theory and polynomial computability. *All Union Conf. Appl. Logic., Proc.* Novosibirsk, 1985, 188–191 (In Russian) Cf. also essentially extended English version with new results in *FCT'87,* LNCS **278** (1987), 391–397
34. V. Sazonov, Hereditarily-finite sets, data bases and polynomial-time computability. *TCS*, **119** Elsevier (1993), 187–214
35. V. Sazonov, A bounded set theory with anti-foundation axiom and inductive definability, *CSL'94* Springer LNCS **933** (1995) 527–541
36. V. Sazonov, On Bounded Set Theory. Invited talk on the *10th International Congress on Logic, Methodology and Philosophy of Sciences, in Volume I: Logic and Scientific Method*, Kluwer Academic Publishers, 1997, 85–103
37. V. Sazonov, Web-like Databases, Anti-Founded Sets and Inductive Definability, *Programmirovanie*, 1999, N5, 26-43 (In Russian; cf. also English version of this J.).
38. V. Sazonov and Yu. Serdyuk, Experimental implementation of set-theoretic query language Delta to Web-like databases (in Russian), *Programmnye Systemy, Teoreticheskie Osnovy i prilozhenija,* RAN, Institut Programmnyh System, Moskva, Nauka, Fizmatlit, 1999
39. Yu. Serdyuk, Partial Evaluation in a Set-Theoretic Query Language for WWW, *FoIKS'2000*, Springer LNCS **1762** (2000), 260–274
40. D. Suciu, Distributed Query Evaluation on Semistructured Data, 1997. Available from `http://www.research.att.com/~suciu`
41. D. Suciu, Query decomposition and view maintenance for query languages for unstructured data, *Proc. of the Internat. Conf. on Very Large Data Bases*, 1996
42. M.Y. Vardi, The complexity of relational query languages. *Proc. of the 14th. ACM Symp. on Theory of Computing*, (1982) 137–146

Reexecution-Based Analysis of Logic Programs with Delay Declarations

Agostino Cortesi[1], Baudouin Le Charlier[2], and Sabina Rossi[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia,
via Torino 155, 30172 Venezia, Italy
{cortesi,srossi}@dsi.unive.it

[2] Universite Catholique de Louvain, Departement d'ingegnierie Informatique
2, Place Sainte-Barbe, B-1348 Louvain-la-Neuve (Belgique)
blc@info.ucl.ac.be

Abstract. A general semantics-based framework for the analysis of logic programs with delay declarations is presented. The framework incorporates well known refinement techniques based on reexecution. The concrete and abstract semantics express both deadlock information and qualified answers.

1 Introduction

In order to get more efficiency, users of current logic programming environments, like Sictus-Prolog [13], Prolog-III, CHIP, SEPIA, etc., are not forced to use the classical Prolog left-to-right scheduling rule. Dynamic scheduling can be applied instead where atom calls are delayed until their arguments are sufficiently instantiated, and procedures are augmented with delay declarations.

The analysis of logic programs with dynamic scheduling was first investigated by Marriott *et al.* in [18,11]. A more general (denotational) semantics of this class of programs, extended to the general case of CLP, has been presented by Falaschi *et al.* in [12], while verification and termination issues have been investigated by Apt and Luitjes in [2] and by Marchiori and Teusink in [17], respectively.

In this paper we discuss an alternative, strictly operational, approach to the definition of concrete and abstract semantics for logic programs with delay declarations.

The main intuitions behind our proposal can be summarized as follows:

- to define in a uniform way concrete, collecting, and abstract semantics, in the spirit of [14]: this allows us to easily derive correctness proofs of the whole analyses;
- to define the analysis as an extension of the framework depicted in [14]: this allows us to reuse existing code for program analysis, with minimal additional effort;
- to explicitly derive deadlock information (possible deadlock and deadlock freeness) producing, as a result of the analysis, an approximation of concrete qualified answers;

- to apply the reexecution technique developed in [15]: if during the execution of an atom a a deadlock occurs, then a is allowed to be reexecuted at a subsequent step.

The main difference between our approach and the ones already presented in the literature is that we are mainly focussed on analysis issues, in particular on deadlock and no-deadlock analysis. This motivates the choice of a strictly operational approach, where deadlock information is explicitly maintained.

In this paper we present an extension of the specification of the GAIA abstract interpreter [14] to deal with dynamic scheduling. We design both a concrete and an abstract semantics, as well as a a generic algorithm that computes a fixpoint of the abstract semantics. This is done following the classical abstract interpretation methodology.

The main idea is partitioning literals of a goal g into three sets: literals which are delayed, literals which are not delayed and have not been executed yet, and literals which are allowed to be reexecuted as they are not delayed but have already been executed before and fallen into deadlock. This partitioning dramatically simplifies both concrete and abstract semantics with respect to the approach depicted in [8], where a preliminary version of this work was presented.

Our approach uses the reexecution technique which exploits the well known property of logic programming that a goal may be reexecuted arbitrarily often without affecting the semantics of the program. This property has been pointed out since 1987 by Bruynooghe [3,4] and subsequently used in abstract interpretation to improve the precision of the analysis [15]. In this framework, reexecution allows to improve the accuracy of deadlock analysis, and its application may be tuned according to computational constraints.

The rest of the paper is organized as follows. In the next section we recall some basic notions about logic programs with delay declarations. Section 3 depicts the concrete operational semantics which serves as a basis for the new abstract semantics introduced in Section 4. Correctness of our generic fixpoint algorithm is discussed. Section 5 concludes the paper.

2 Logic Programs with Delay Declarations

Logic programs with delay declarations consist of two parts: a logic program and a set of delay declarations, one for each of its predicate symbols.

A *delay declaration* associated for an n-ary predicate symbol p has the form

$$\texttt{DELAY}\ \ p(x_1,\ldots,x_n)\ \ \texttt{UNTIL}\ \ \mathit{Cond}(x_1,\ldots,x_n)$$

where $\mathit{Cond}(x_1,\ldots,x_n)$ is a formula in some assertion language. We are not concerned here with the syntax of this language since it is irrelevant for our purposes. The meaning of such a delay declaration is that an atom $p(t_1,\ldots,t_n)$ can be selected in a query only if the condition $\mathit{Cond}(t_1,\ldots,t_n)$ is satisfied. In this case we say that the atom $p(t_1,\ldots,t_n)$ *satisfies* its delay declaration.

A derivation of a program augmented with delay declarations *succeeds* if it ends with the empty goal; while it *deadlocks* if it ends with a non-empty goal no atom of which satisfies its delay declaration. Both successful and deadlocked derivations compute *qualified answers*, i.e., pairs of the form $\langle\theta, d\rangle$ where d is the last goal (that is a possibly empty sequence of delayed atoms) and θ is the substitution obtained by concatenating the computed mgu's from the initial goal. Notice that, if $\langle\theta, d\rangle$ is a qualified answer for a successful derivation then d is the empty goal and θ restricted to the variables of the initial goal is the corresponding computed answer substitution. We denote by $qans_P(g)$ the set of qualified answers for a goal g and a program P.

We restrict our attention to delay declarations which are *closed under instantiation*, i.e., if an atom satisfies its delay declaration then also all its instances do. Notice that this is the choice of most of the logic programming systems dealing with delay declarations such as IC-Prolog, NU-Prolog, Prolog-II, Sicstus-Prolog, Prolog-III, CHIP, Prolog M, SEPIA, etc.

The following example illustrates the use of delay declarations in logic programming.

Example 1. Consider the program PERMUTE discussed by Naish in [19].

```
% perm(Xs,Ys) ← Ys is a permutation of the list Xs
  perm(Xs,Ys) ← Xs = [ ], Ys = [ ].
  perm(Xs,Ys) ← Xs = [X|X1s], delete(X,Ys,Zs), perm(X1s,Zs).

% delete(X,Ys,Zs) ← Zs is the list obtained by removing X from the list Ys
  delete(X,Ys,Zs) ← Ys = [X|Zs].
  delete(X,Ys,Zs) ← Ys = [X1|Y1s], Zs = [X1|Z1s], delete(X,Y1s,Z1s).
```

Clearly, the relation declaratively given by perm is symmetric. Unfortunately, the behavior of the program with Prolog (using the leftmost selection rule) is not. In fact, given the query

$$Q_1 := \leftarrow \texttt{perm}(\texttt{Xs}, [\texttt{a}, \texttt{b}]).$$

Prolog will correctly backtrack through the answers $\texttt{Xs} = [\texttt{a}, \texttt{b}]$ and $\texttt{Xs} = [\texttt{b}, \texttt{a}]$. However, for the query

$$Q_2 := \leftarrow \texttt{perm}([\texttt{a}, \texttt{b}], \texttt{Xs}).$$

Prolog will first return the answer $\texttt{Xs} = [\texttt{a}, \texttt{b}]$ and on subsequent backtracking will fall into an infinite derivation without returning answers anymore.

For languages with delay declarations the program PERMUTE behaves symmetrically. In particular, if we consider the delay declarations:

```
DELAY perm(Xs,_) UNTIL nonvar(Xs).
DELAY delete(_,_,Zs) UNTIL nonvar(Zs).
```

the query Q_2 above does not fall into a deadlock. ∎

Under the assumption that delay declarations are closed under instantiation, the following result, which is a variant of Theorem 4 in Yelick and Zachary [21], holds.

$P \in \mathit{Programs}$	$P ::= pr_1, \dots, pr_n \ (n > 0)$
$pr \in \mathit{Procedures}$	$pr ::= c_1, \dots, c_n \ (n > 0)$
$c \in \mathit{Clauses}$	$c ::= h : -g.$
$h \in \mathit{ClauseHeads}$	$h ::= p(x_1, \dots, x_n) \ (n \geq 0)$
$g \in \mathit{LiteralSequences}$	$g ::= l_1, \dots, l_n \ (n \geq 0)$
$l \in \mathit{Literals}$	$l ::= a \mid b$
$a \in \mathit{Atoms}$	$a ::= p(x_{i_1}, \dots, x_{i_n}) \ (n \geq 0)$
$b \in \mathit{Built\text{-}ins}$	$b ::= x_i = x_j \mid x_{i_1} = f(x_{i_2}, \dots, x_{i_n})$
$p \in \mathit{ProcedureNames}$	
$f \in \mathit{Functors}$	
$x_i \in \mathit{ProgramVariables}$	

Fig. 1. Abstract syntax of normalized programs

Theorem 1. *Let P be a program augmented with delay declarations, g be a goal and g' be a permutation of g. Then $qans_P(g)$ and $qans_P(g')$ are equals modulo the ordering of delayed atoms.*

It follows that both successful and deadlocked derivations are "independent" from the choice of the selection rule. Moreover, Theorem 1 allows us to treat goals as multisets instead of sequences of atoms.

3 The Concrete Operational Semantics

In this section we describe a concrete operational semantics for pure Prolog augmented with delay declarations. The concrete semantics is the link between the standard semantics of the language and the abstract one. We assume a preliminary knowledge of logic programming (see, [1,16]).

3.1 Programs and Substitutions

Programs are assumed to be normalized according to the syntax given in Fig. 1. The variables occurring in a literal are distinct; distinct procedures have distinct names; all clauses of a procedure have exactly the same head; if a clause uses m different program variables, these variables are $x_1, \dots, x_m$. If $g := a_1, \dots, a_n$ we denote by $g \setminus a_i$ the goal $g' := a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$.

We assume the existence of two disjoint and infinite sets of variables: *program variables*, which are ordered and denoted by x_1, x_2, ..., x_i, ..., and *standard variables* which are denoted by letters y and z (possibly subscripted). Programs are built using program variables only.

A program substitution is a set $\{x_{i_1}/t_1, \dots, x_{i_n}/t_n\}$, where $x_{i_1}, \dots, x_{i_n}$ are distinct program variables and $t_1, \dots, t_n$ are terms (built with standard variables only). Program substitutions are not substitutions in the usual sense; they are best understood as a form of program store which expresses the state of the computation at a given program point. It is meaningless to compose them as usual substitutions. The domain of a program substitution $\theta = \{x_{i_1}/t_1, \dots, x_{i_n}/t_n\}$, denoted by $dom(\theta)$, is the set of program variables $\{x_{i_1}, \ldots, x_{i_n}\}$. The application $x_i\theta$ of a program substitution θ to a program variable x_i is defined only if $x_i \in dom(\theta)$: it denotes the term bound to x_i in θ. Let D be a finite set of program variables. We denote by PS_D the set of program substitutions whose domain is D.

3.2 Concrete Behaviors

The notion of concrete behavior provides a mathematical model for the input/output behavior of programs. To simplify the presentation, we do not parameterize the semantics with respect to programs. Instead, we assume given a fixed underlying program P augmented with delay declarations.

We define a *concrete behavior* as a relation from input states to output states as defined below. The *input states* have the form

- $\langle \theta, p \rangle$, where p is the name of a procedure and θ is a program substitution also called activation substitution. Moreover, $\theta \in PS_{\{x_1,\ldots,x_n\}}$, where $x_1, \ldots, x_n$ are the variables occurring in the head of every clause of p.

The *output states* have the form

- $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and κ is a deadlock state, i.e., it is an element from the set $\{\delta, \nu\}$, where δ stands for *definite deadlock*, while ν stands for *no deadlock*. In case of no deadlock, θ' restricted to the variables $\{x_1, \ldots, x_n\}$ is a computed answer substitution (the one corresponding to a successful derivation), while in case of deadlock, θ' is the substitution part of a qualified answer to p and coincides with a partial answer substitution for it.

We use the relation symbol $\longmapsto$ to represent concrete behaviors, i.e., we write $\langle \theta, p \rangle \longmapsto \langle \theta', \kappa \rangle$: this notation emphasizes the similarities between this concrete semantics and the structural operational semantics for logic programs defined in [15]. Concrete behaviors are intended to model successful and deadlocked derivations of atomic queries.

3.3 Concrete Semantic Rules

The concrete semantics of an underlying program P with delay declarations is the least fixpoint of a continuous transformation on the set of concrete behaviors. This transformation is defined in terms of semantic rules that naturally extend concrete behaviors in order to deal with clauses and goals. In particular, a concrete behavior is extended through intermediate states of the form $\langle \theta, c \rangle$ and $\langle \theta, g\_d, g\_e, g\_r \rangle$, where c is a clause and g_d, g_e, g_r is a partition of a goal g such that: g_d contains all literals in g which are delayed, g_e contains all literals in g which are not delayed and have not been executed yet, g_r contains all literals in g which are allowed to be reexecuted, i.e., all literals that are not delayed and have already been executed but fallen into a deadlock.

– Each pair $\langle \theta, c \rangle$, where c is a clause, $\theta \in PS_{\{x_1,\ldots,x_n\}}$ and $x_1, \ldots, x_n$ are the variables occurring in the head of c, is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state;

– Each tuple $\langle \theta, g\_d, g\_e, g\_r \rangle$, where $\theta \in PS_{\{x_1,\ldots,x_m\}}$ and $x_1, \ldots, x_m$ are the variables occurring in (g_d, g_e, g_r), is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_m\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state.

We briefly recall here the concrete operations which are used in the definition of the concrete semantic rules depicted in Fig. 2. The reader may refer to [14] for a complete description of all operations but the last one, `SPLIT`, that is brand new.

- `EXTC` is used at clause entry: it extends a substitution on the set of variables occurring in the body of the clause.
- `RESTRC` is used at clause exit: it restricts a substitution on the set of variables occurring in the head of the clause.
- `RETRG` is used when a literal l occurring in the body of a clause is analyzed. Let $\{x_{i_1}, \ldots, x_{i_n}\}$ be the set of variables occurring in l. This operation expresses a substitution on variables $x_{i_1}, \ldots, x_{i_n}$ in terms of the formal parameters $x_1, \ldots, x_n$.
- `EXTG` is used to combine the analysis of a built-in or a procedure call (expressed in terms of the formal parameters $x_1, \ldots, x_n$) with the activating substitution.
- `UNIF-FUNC` and `UNIF-VAR` are the operations that actually perform the unification of equations of the form $x_i = x_j$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$, respectively.
- `SPLIT` is a new operation: given a substitution θ and a goal g, it partitions g into the set of atoms g_d which do not satisfy the corresponding delay declarations, and then are not executable, and the set of atoms g_e which satisfy the corresponding delay declarations, and then are executable.

The definition of the concrete semantic rules proceeds by induction on the syntactic structure of program P. Rule $\mathbf{R}_1$ defines the result of executing a procedure call: this is obtained by executing any clause defining it. Rule $\mathbf{R}_2$ defines the result of executing a clause: this is obtained by executing its body under the same input substitution after splitting the body into two parts: executable literals and delayed literals. Rule $\mathbf{R}_3$ defines the result of executing the empty goal, generating a successful output substitution. Rule $\mathbf{R}_4$ defines a deadlock situation that yields a definite deadlock information δ. Rules $\mathbf{R}_5$ to $\mathbf{R}_8$ specify the execution of a literal. First, the literal is executed producing an output substitution θ_3; then reexecutable atoms are (re)executed through the auxiliary relation $\langle \theta_3, g\_r \rangle \longmapsto_r \langle \theta_4, \bar{g}\_r \rangle$: its effect is to refine θ_3 into θ_4 and to remove from g_r the atoms that are completely solved in θ_4 returning the new list of reexecutable atoms $\bar{g}\_r$; finally, the sequence of delayed atoms with the new substitution θ_4 is partitioned in two sets: the atoms that are still delayed and those that have been awakened. Rules $\mathbf{R}_5$ and $\mathbf{R}_6$ specify the execution of built-ins and use the unification operations. Rules $\mathbf{R}_7$ and $\mathbf{R}_8$ define the execution of an atom a

The reexecutable rules defining the auxiliary relation $\longmapsto_r$ can be easily obtained according to the methodology in [15].

The concrete semantics of a program P with delay declarations is defined as a fixpoint of this transition system. We can prove that this operational semantics is safe with respect to the standard resolution of programs with delay declarations.

$$\textbf{R1}\;\frac{\begin{array}{c} c \text{ is a clause defining } p \\ \langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, p\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R2}\;\frac{\begin{array}{c} c := h :- g \\ \theta_1 = \mathtt{EXTC}(c, \theta) \\ \langle g\_d, g\_e\rangle = \mathtt{SPLIT}(\theta_1, g) \\ \langle \theta_1, g\_d, g\_e, < > \rangle \longmapsto \langle \theta_2, \kappa\rangle \\ \theta' = \mathtt{RESTRC}(c, \theta_2) \end{array}}{\langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R3}\;\frac{}{\langle \theta, < >, < >< >, \rangle \longmapsto \langle \theta, \nu\rangle} \qquad \textbf{R4}\;\frac{\text{either } g\_d \neq < > \text{ or } g\_r \neq < >}{\langle \theta, g\_d, < >, g\_r\rangle \longmapsto \langle \theta, \delta\rangle}$$

$$\textbf{R5}\;\frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus b \\ b := x_i = x_j \\ \theta_1 = \mathtt{RESTRG}(b, \theta) \\ \theta_2 = \mathtt{UNIF\_VAR}(\theta_1) \\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R6}\;\frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus b \\ b := x_i = f(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(b, \theta) \\ \theta_2 = \mathtt{UNIF\_FUNC}(b, \theta_1) \\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R7}\;\frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \nu\rangle \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R8}\;\frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \delta\rangle \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g\_r.a\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

Fig. 2. Concrete semantic rules

4 Collecting and Abstract Semantics

As usual in the *Abstract Interpretation* approach [9,10], in order to define an abstract semantics we proceed in three steps. First, we depict a collecting semantics, by lifting the concrete semantics to deal with sets of substitutions. Then, any abstract semantics will be defined as an abstraction of the collecting semantics: it is sufficient to provide an abstract domain that enjoys a Galois connection with the concrete domain $\wp(Subst)$, and a suite of abstract operations that safely approximate the concrete ones. Finally, we draw an algorithm to compute a (post-)fixpoint of an abstract semantics defined this way.

The collecting semantics can be trivially obtained from the concrete one by

- replacing substitutions with sets of substitutions;
- using μ, standing for *possible deadlock*, instead of δ;
- redefining all operations in order to deal with sets of substitutions (as done in [14]).

In particular, the collecting version of operation `SPLIT`, given a set of substitutions Θ, will partition a goal g into the set of atoms g_d which do not satisfy the corresponding delay declarations for some $\theta \in \Theta$, and the set of atoms g_e which do satisfy the corresponding delay declarations for some $\theta \in \Theta$. Notice that this approach is sound, i.e., if an atom is executed at the concrete level then it will be also at the abstract level. However, since some atoms can be put both in g_d and in g_e some level of imprecision could arise.

Once the collecting semantics is fixed, deriving abstract semantics is almost an easy job. Any domain abstracting substitutions can be used to describe abstract activation states. Similarly to the concrete case, we distinguish among input states, e.g., $\langle \beta, p \rangle$ where β is an approximation of a set of activation substitutions, and output states, e.g., $\langle \beta', \kappa \rangle$ where β' is an approximation of a set of output substitutions and $\kappa \in \{\mu, \nu\}$ is an abstract deadlock state. Clearly, the accuracy of deadlock analysis will depend on the matching between delay declarations and the information represented by the abstract domains. It is easy to understand, by looking at the concrete semantics presented above, that very few additional operations should be implemented on an abstract substitution domain like the ones in [6,7,14], while a great amount of existing specification and coding can be reused for free.

Fig. 3 reports the final step in the Abstract Interpretation picture described above: an abstract transformation that abstracts the concrete semantics rules. The abstract semantics is defined as a post-fixpoint of transformation TAB on sets of abstract tuples, sat, as defined in the picture. An algorithm computing the abstract semantics can be defined by simple modification of the reexecution fixpoint algorithm presented in [15]. The reexecution function T_r is in the spirit of [15]. It uses the abstract operations `REFINE` and `RENAME`, where

- `REFINE` is used to refine the result β of executing an atom by combining it with the results obtained by reexecution of atoms in the reexecutable atom lists starting from β itself;
- `RENAME` is used after reexecution of an atom a: it expresses the result of reexecution in terms of the variables $x_{i_1}, \ldots, x_{i_n}$ occurring in a.

```
$TAB(sat) = \{(\beta, p, \langle\beta', \kappa\rangle) : (\beta, p)$ is an *input state* and $\langle\beta', \kappa\rangle = T_p(\beta, p, sat)\}$.

$T_p(\beta, p, sat) = \texttt{UNION}(\langle\beta_1, \kappa_1\rangle \ldots, \langle\beta_n, \kappa_n\rangle)$
    **where**  $\langle\beta_i, \kappa_i\rangle = T_c(\beta, c_i, sat)$,
               $c_1, \ldots, c_n$ are the clauses defining $p$.

$T_c(\beta, c, sat) = \langle\texttt{RESTRC}(c, \beta'), \kappa\rangle$
    **where**  $\langle\beta', \kappa\rangle = T_b(\texttt{EXTC}(c, \beta), g\_d, g\_e, < >, sat)$,
               $\langle g\_d, g\_e\rangle = \texttt{SPLIT}(\beta, b)$ where $b$ is the body of $c$.

$T_b(\beta, < >, < >, < >, sat) = \langle\beta, \nu\rangle$.

$T_b(\beta, g\_d, < >, g\_r, sat) = \langle\beta, \mu\rangle$
    **where**  either $g\_d$ or $g\_r$ is not empty.

$T_b(\beta, g\_d, l.g\_e, g\_r, sat) = T_b(\beta_4, \bar{g}\_d, g\_e.\bar{g}\_e, \bar{g}\_r, sat)$
    **where**  $\langle\bar{g}\_d, \bar{g}\_e\rangle = \texttt{SPLIT}(\beta_4, g\_d)$
               $\langle\beta_4, \bar{g}\_r\rangle = T_r(\beta_3, g\_r, sat)$           if $\kappa = \nu$,
                                $T_r(\beta_3, g\_r.l, sat)$           if $\kappa = \mu$,
               $\beta_3 = \texttt{EXTG}(l, \beta, \beta_2)$,
               $\langle\beta_2, \kappa\rangle = sat(\beta_1, p)$                    if $l$ is $p(\cdots)$
                            $\langle\texttt{UNIF\_VAR}(\beta_1), \nu\rangle$          if $l$ is $x_i = x_j$,
                            $\langle\texttt{UNIF\_FUNC}(l, \beta_1), \nu\rangle$      if $l$ is $x_i = f(\cdots)$,
               $\beta_1 = \texttt{RESTRG}(l, \beta)$.

$T_r(\beta, (a_1, \ldots, a_n), sat) = \sqcap_{i=1}^{\infty}\langle\beta_i, g_i\rangle$
    **where**  $\langle\beta_0, g_0\rangle = \langle\beta, (a_1, \ldots, a_n)\rangle$
               $\beta_{i+1} = \texttt{REFINE}(\beta_i, T_r(\beta_i, a_1, sat), \ldots, T_r(\beta_i, a_n, sat))$ $(i \geq 1)$
               $g_{i+1} = \{a_i \mid i \in \{1, \ldots, n\}$ and $\langle\bullet, \mu\rangle = T_r(\beta_i, a_i, sat)\}$

$T_r(\beta, a, sat) = \langle\texttt{RENAME}(a, \beta_2), \kappa\rangle$
    **where**  $\langle\beta_2, k\rangle = sat(\beta_1, p)$                          if $a$ is $p(\cdots)$
               $\beta_1 = \texttt{RESTRG}(a, \beta)$.
```

Fig. 3. The abstract transformation

As already observed before, most of the operations that are used in the algorithm are simply inherited from the GAIA framework [14]. The only exception is SPLIT, which depends on a given set of delay declarations.

The correctness of the algorithm can be proven the same way as in [14] and [15]. What about termination ? The execution of T_b terminates since the number of literals in g_d and g_e decreases of exactly one at each recursive call. The fact that the execution of T_r terminates depends on some hypothesis on the abstract domain such as to be a complete lattice (when this is not the case, and it is just a cpo, an additional widening operation is usually provided by the domain).

Example 2. Consider again the program `PERMUTE` illustrated above. Using one of our domains for abstract substitutions, like `Pattern` (see [5,20]), and starting from an activation state of the form `perm(ground,var)` our analysis returns the abstract qualified answer $\langle \texttt{perm}(\texttt{ground}, \texttt{ground}), \nu \rangle$, which provides the information that any concrete execution, starting in a query of `perm` with the first argument being ground and the second one being variable, is deadlock free.

5 Conclusions

The framework presented in this paper is part of a project aimed at integrating most of the work, both theoretical and practical, on abstract interpretation of logic programs developed by the authors in the last years. The final goal is to get a practical tool that tackles a variety of problems raised by the recent research and development directions in declarative programming. Dynamic scheduling is an interesting example in that respect, as most of current logic programming environments integrate this feature.

In the next future, we plan to adapt the existing implementations of GAIA systems in order to practically evaluate the accuracy and efficiency of the this framework.

Acknowledgments. This work has been partially supported by the Italian MURST Projects “Interpretazione Astratta, Type Systems e Analisi Control-Flow”, and “Certificazione automatica di programmi mediante interpretazione astratta”.

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. *Lecture Notes in Computer Science*, 936:66–80, 1995.
3. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
4. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, California, August 1987. Computer Society Press of the IEEE.
5. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, May 1996.
6. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming. In *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
7. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 28(1–3):27–71, 2000.
8. A. Cortesi, S. Rossi, and B. Le Charlier. Operational semantics for reexecution-based analysis of logic programs with delay declarations. *Electronic Notes in Theoretical Computer Science*, 48(1), 2001. http://www.elsevier.nl/locate/entcs.

9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, pages 269–282, Los Angeles, California, January 1979.
11. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. MIT Press, 1995.
12. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Information and Computation*, 137(1):41–67, 1997.
13. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1998. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.
14. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.
15. B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
16. J.W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.
17. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 447–464, Cambridge, December 4–7 1995. MIT Press.
18. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.
19. L. Naish. *Negation and control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.
20. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.
21. K. Yelick and J. Zachary. Moded type systems for logic programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 116–124, 1989.

Pos($\mathcal{T}$): Analyzing Dependencies in Typed Logic Programs

Maurice Bruynooghe[1], Wim Vanhoof[1], and Michael Codish[2]

[1] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{maurice,wimvh}@cs.kuleuven.ac.be
[2] Ben-Gurion University, Department of Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel; mcodish@cs.bgu.ac.il

Abstract. Dependencies play a major role in the analysis of program properties. The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is a main applications area. The precision of such an analysis can be improved through the integration of either pattern information or type information. This paper develops an approach in which type information is exploited. Different from previous work, a separate simple analysis is performed for each subtype of the types. If the types are polymorphic, then dependencies for specific type instances are derived from the analysis of the polymorphic typed program.

1 Introduction

Dependencies play an important role in program analysis. A statement "program variable X has property p" can be represented by the propositional variable x^p and dependencies between properties of program variables can be captured as Boolean functions. For example, the function denoted by $x^p \rightarrow y^p$ specifies that whenever x has property p then so does y. In many cases, the precision of a dataflow analysis for a property p is improved if the underlying analysis domain captures dependencies with respect to that given property. The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is a main applications area. It aims at identifying which program variables are ground i.e. cannot be further instantiated. The class of *positive* Boolean functions, *Pos* consists of the Boolean functions f for which $f(true, \ldots, true) = true$.

One of the key steps in a groundness dependency analysis is to characterise the dependencies imposed by the unifications that could occur during execution. If the program specifies a unification of the form $term_1 = term_2$ and the variables in $term_1$ and $term_2$ are $\{X_1, \ldots, X_m\}$ and $\{Y_1, \ldots, Y_n\}$ respectively, then the corresponding groundness dependency imposed is $(x_1 \wedge \cdots \wedge x_m) \leftrightarrow (y_1 \wedge \cdots \wedge y_n)$ where the groundness of a program variable X is represented by the propositional variable x. This formula specifies that variables in $term_1$ are (or will become) ground if and only if the variables in $term_2$ are (or do). It is possible to improve the precision of an analysis if additional information about the structure (or patterns) of terms is available. For example, if we know that $term_1$ and $term_2$ are both difference lists of the form $H_1 - T_1$ and $H_2 - T_2$, respectively, then the unification $term_1 = term_2$ imposes the dependency $(h_1 \leftrightarrow h_2) \wedge (t_1 \leftrightarrow t_2)$ which is more precise than $(h_1 \wedge t_1) \leftrightarrow (h_2 \wedge t_2)$ which would be derived without the additional information. This has been the approach in previous works such as [20,24,8,11,2] where pattern analysis can be used to enhance the precision of other analyses.

Introducing pattern information does not allow to distinguish between e.g. bounded lists such as $[1, X, 3]$ and open ended lists such as $[1, 2|Z]$ because the open end can be situated at an arbitrary depth. Making such distinction requires to consider type information. The domain Pos has been adapted to do so in [6] where each type was associated with an incarnation of Pos. However, that analysis was for untyped programs and each incarnation was developed in a somewhat ad-hoc fashion. The model based analysis of [14] is also related. Here, the models of the program, based on different pre-interpretations express different kinds of program properties. But again, the choice of a pre-interpretation is on a case by case basis. Others in one or another way annotate the types with information about the positions where a variable can occur. This is the case in: [26,27]; the binding time analysis of [30]; and also in [21] which uses types and applies linear refinement to enrich the type domain with Pos-like dependencies. Most close to our approach is the work of [19] which associates properties with what we call in this paper the constituents of the type of the variable (the types possible for subterms of its value). In that work, unifications are abstracted by Boolean formulas expressing the groundness dependencies between different constituents. The main difference is that they construct a single compiled clause covering all constituents while we construct one clause for each constituent. A feature distinguishing our work from the other type based approaches is that we have an analysis for polymorphic types which eliminates the need to analyze a polymorphic predicate for each distinct type instance under which it is called.

Type information can be derived by analysis, as e.g. in [18,15]; specified by the user and verified by analysis as possible in Prolog systems such as Ciao [16]; or declared and considered part of the semantics of the program as with strongly typed languages such as Gödel [17], Mercury [28] and HAL [13]. The analysis as worked out in this paper is for strongly typed programs.

The next section recalls the essentials of Pos. Section 3 introduces types and defines the constituents of a type. Section 4 develops the analysis for monomorphic types, it defines the abstraction for different kinds of unification and proves that they are correct. Section 5 deals with the polymorphic case. It describes how to abstract a call with types that are an instance of the polymorphic types in the called predicate's definition. It proves that the results of the polymorphic analysis approximate the results of a monomorphic analysis and points out the (frequent) cases where both analyses are equivalent. Section 6 discusses applications and related work.

2 Analyzing Groundness Dependencies with *P*os

Program analysis aims at computing finite approximations of the possibly infinite number of program states that could arise at runtime. Using abstract interpretation [12], approximations are expressed using elements of an abstract domain and are computed by abstracting a concrete semantics; the algebraic properties of the abstract domain guarantee that the analysis is terminating and correct.

The formal definition of *Pos* states that a *P*os function φ describes a substitution θ (a program state) if any set of variables that might become ground by further instantiating θ is a model of φ. For example, the models of $\varphi = x \wedge (y \rightarrow z)$ are $\{\{X\}, \{X, Z\}, \{X, Y, Z\}\}$. We can see that φ describes $\theta = \{X/a, Y/f(U, V)), Z/g(U)\}$ because under further instantiation X is in all of the models and if Y is in a model (becomes ground) then so is Z. Notice that $\theta = \{X/a\}$ is not described by φ as $\{X/a, Y/a\}$ is a further instantiation of θ and $\{X, Y\}$ is not a model of φ. *Pos* satisfies the algebraic properties required of an abstract domain [9]. See also [23] for more details.

A simple way of implementing a *P*os based groundness analysis is described in [7] and is illustrated in Figure 1. For the purpose of this paper it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Figure 1) is reduced to the problem of computing the concrete semantics of the abstract program (in the middle and on the right). The result is given at the bottom of the figure. For additional details of why this is so, refer to [7, 10,23]. The least model of the abstract program (e.g. computed using meta-interpreters such as those described in [5,7]) is interpreted as representing the propositional formula $x_1 \leftrightarrow x_2$ and $(x_1 \wedge x_2) \leftrightarrow x_3$ for the atoms `rotate(X1,X2)` and `append(X1,X2,X3)` respectively. This illustrates a goal-independent analysis. Goal-dependent analyses are supported by applying Magic sets or similar techniques (see e.g. [7]).

Concrete `rotate`	*Abstract* `rotate`	*Auxiliary predicates*
`rotate(Xs,Ys) :-`	`rotate(Xs,Ys) :-`	`iff(true,[]).`
`  append(As,Bs,Xs),`	`  append(As,Bs,Xs),`	`iff(true,[true|Xs]) :-`
`  append(Bs,As,Ys).`	`  append(Bs,As,Ys).`	`  iff(true,Xs).`
		`iff(false,Xs) :-`
`append(Xs,Ys,Zs) :-`	`append(Xs,Ys,Zs) :-`	`  member(false,Xs).`
`  Xs = [],`	`  iff(Xs,[]),`	
`  Ys = Zs.`	`  iff(Ys,[Zs]).`	
`append(Xs,Ys,Zs) :-`	`append(Xs,Ys,Zs) :-`	
`  Xs = [X|Xs1],`	`  iff(Xs,[X,Xs1]),`	
`  Zs = [X|Zs1],`	`  iff(Zs,[X,Zs1]),`	
`  append(Xs1,Ys,Zs1).`	`  append(Xs1,Ys,Zs1).`	
	`rotate(X,X).`	`append(true,true,true).`
Least model (abstract `rotate`*):*		`append(false,Y,false).`
		`append(X,false,false).`

Fig. 1. Concrete and abstract programs; least model of the abstract program

3 About Terms and Types

We assume familiarity with logic programming concepts [22,1]. We let *Term* denote the set $T(\Sigma, V)$ of terms constructed from the sets of function symbols Σ and variables V. Types, like terms are constructed from (type) variables and (type) symbols. We denote by $\mathcal{T}$ the set of terms $T(\Sigma_\mathcal{T}, V_\mathcal{T})$ (or types in this case) constructed from type symbols $\Sigma_\mathcal{T}$ and type variables $V_\mathcal{T}$. We assume that the sets of symbols, variables, type symbols and type variables are fixed and disjoint. We write $t{:}\tau$ to specify that term t has type τ and f/n to specify that (function or type) symbol f has arity n. Besides the usual substitutions which are mappings from variables to terms, we also have type substitutions which are mappings from type variables to types.

We assume a standard notion of strong typing as for example in Mercury [28]. This means that each predicate is associated with a single type declaration and that programs are well typed. Hence, one can associate a unique type with every term and variable in a program clause. A type containing type variables is said to be *polymorphic*, otherwise it is *monomorphic*. The application of a type substitution to a polymorphic type gives a new type which is an *instance* of the original type. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols.

Definition 1. *The rule for a type symbol $h/n \in \Sigma_\mathcal{T}$ is a definition of the form $h(\bar{V}) \longrightarrow f_1(\bar{\tau}_1)\ ; \dots\ ;\ f_k(\bar{\tau}_k)$. where: $\bar{V}$ is an n-tuple from $V_\mathcal{T}$; for $1 \le i \le k$, $f_i/m \in \Sigma$ with $\bar{\tau}_i$ an m-tuple from $\mathcal{T}$; and type variables occurring in the right hand side occur in $\bar{V}$. The function symbols $\{f_1, \dots, f_k\}$ are said to be associated with the type symbol h. A finite set of type rules is called a type definition.*

Example 1. The keyword `type` introduces a type rule:

```
type list(T) ---> [] ; [T | list(T)].
```

The function symbols $[\cdot]$ (nil) and $[\cdot|\cdot]$ (cons) are associated with the type symbol *list*. The type definition specifies also the denotation of each type (the set of terms belonging to the type). For this example, terms of polymorphic type $list(T)$ are variables (typed $list(T)$), terms of the form $[\,]$, or terms of the form $[t_1|t_2]$ with t_1 of type T and t_2 of type $list(T)$. As T is a type variable we cannot determine or commit to its structure under instantiation. Hence only a variable can be of type T. Applying the type substitution $\{T/int\}$ on $list(T)$ gives the type $list(int)$. Terms of the form $[t_1|t_2]$ are of type $list(int)$ if t_1 is of type int and t_2 is of type $list(int)$. Type instances can also be polymorphic, e.g. $list(list(T))$.

The next definition specifies the *constituents* of a type τ. These are the possible types for subterms of terms of type τ.

Definition 2. *Let $h(\bar{\tau}) \longrightarrow f_1(\bar{\tau}_1)\ ; \dots\ ;\ f_k(\bar{\tau}_k)$ be an instance of a rule in ρ and τ an argument of one of the $f_i(\bar{\tau}_i)$. We say that τ is a ρ-constituent of $h(\bar{\tau})$. The constituents relation is the minimal pre-order (reflexive and transitive) $\preceq_\rho$: $\mathcal{T} \times \mathcal{T}$ including all pairs $\tau' \preceq_\rho \tau$ such that τ' is a ρ-constituent of τ. The set of ρ-constituents of τ is $Constituents_\rho(\tau) = \left\{ \tau' \in \mathcal{T} \,\middle|\, \tau' \preceq_\rho \tau \right\}$. When clear from the context we omit ρ and write $\preceq$ and Constituents.*

Example 2. With $list/1$ as in Example 1 and the atomic type int, we have:

- $T \preceq list(T)$, $list(T) \preceq list(T)$, $int \preceq list(int)$, $int \preceq int$, $list(T) \preceq list(list(T))$, $T \preceq list(list(T))$, $list(list(T)) \preceq list(list(T))$, $T \preceq T$, ...
- $Constituents(int) = \{int\}$, $Constituents(T) = \{T\}$, $Constituents(list(T)) = \{T, list(T)\}$, $Constituents(list(int)) = \{int, list(int)\}$.

To ensure that analyses are finite, we restrict types to have finite sets of constituents. This is common as a type with an infinite set of constituents is not well defined. For example, consider the definition: `type t(T) ---> f(t(t(T))).` The type $t(T)$ is defined in terms of its constituent type $t(t(T))$, which in turn is defined in terms of its constituent type $t(t(t(T)))$, and so on.

The following defines the instantiation of terms with respect to a given type.

Definition 3. *Let ρ define the types τ and τ'. We say that the term $t{:}\tau'$ is τ-instantiated (under ρ) if: (1) τ is a constituent of τ'; and (2) there does not exist a well-typed instance $t\sigma{:}\tau'$ containing a variable of type τ. The predicate $\mu_\tau^\rho(t{:}\tau')$ is true if and only if $t{:}\tau'$ is τ-instantiated. For a set of typed variables of interest V and a well typed substitution θ, we denote by $\mu_\tau(\theta, V) = \left\{ X{:}\tau' \in V \,\middle|\, \mu_\tau(X\theta{:}\tau') \right\}$. This is the subset of the variables of interest which are τ-instantiated by θ.*

$t{:}list(int)$	$\mu_{list(int)}(t)$	$\mu_{int}(t)$
$[1,2]$	*true*	*true*
$[1,X]$	*true*	*false*
$[1\|X]$	*false*	*false*
$[\,]$	*true*	*true*

Fig. 2. The τ-instantiation of some $list(int)$ terms

If τ' has no constituents of type τ then we do not consider $t{:}\tau'$ as τ-instantiated because no instance of any term of type τ' can have a subterm of type τ. In the case $t{:}T$ where T is a polymorphic parameter (and T has no constituents of type τ), the intuition is that for some instance τ' of T, it could be the case that $t{:}\tau'$ should not be considered τ-instantiated. Figure 2 illustrates the $list(int)$-instantiation and int-instantiation for several terms of type $list(int)$. First note that both $list(int)$ and int are constituents of $list(int)$. Hence, $\mu_{list(int)}([1,X])$ is *true* because all *list(int)*-subterms of well-typed instances of $[1,X]$ are instantiated; and $\mu_{list(int)}([1|X])$ is *false* because the subterm $X{:}list(int)$ is a variable. Also $\mu_{int}([1|X])$ is *false* as e.g. $[1,Y|Z]$ is an instance with the variable Y of type *int*. Note that $\mu_{int}(s) \rightarrow \mu_{list(int)}(s)$. In general, the following property holds:

Proposition 1. *For constituent types $\tau_1 \preceq \tau_2$ and any term t, $\mu_{\tau_1}(t) \rightarrow \mu_{\tau_2}(t)$.*

We observe that classical groundness for a typed term $t{:}\tau$ can be expressed with respect to types instantiation by requiring t to be instantiated with respect to all constituents of τ.

$$ground(t) \leftrightarrow \wedge \left\{ \mu_{\tau_i}^\rho(t{:}\tau) \,\middle|\, \tau_i \in Constituents(\tau) \right\}. \tag{1}$$

4 *Pos*($\mathcal{T}$) in a Monomorphic Setting

We let the propositional variable x^τ denote the predicate $\mu^\rho_\tau(X{:}\tau')$ meaning that typed program variable $X{:}\tau'$ is instantiated with respect to type τ. Instantiation analysis using $Pos(\mathcal{T})$ generalises groundness analysis with Pos. Informally, a positive Boolean formula φ, representing τ-instantiation dependencies, describes a substitution θ if any set of variables that might become τ-instantiated by further instantiating θ is a model of φ. The abstraction and concretization functions are formalised as:

Definition 4. *Let V be a set of (typed) variables of interest and τ a type. The τ-abstraction (in terms of models) of the set Θ of well-typed substitutions and the τ-concretization for the positive Boolean function ψ are:*

$$\alpha_\tau(\Theta) = \left\{\ \mu_\tau(\theta\sigma) \,\middle|\, \theta \in \Theta,\ \sigma \textit{ is a well typed substitution}\ \right\}$$
$$\gamma_\tau(\psi) = \left\{\ \theta \,\middle|\, \alpha_\tau(\{\theta\}) \subseteq \psi\ \right\}$$

Example 3. Let $V = \big\{ X{:}list(int), Y{:}int, Z{:}list(int) \big\}$ and $\theta = \big\{ X \mapsto [Y|Z] \big\}$. Then, $\alpha_{list(int)}(\{\theta\})$ has models $\big\{ \emptyset, \{x,z\} \big\}$ (observe that $Y{:}int$ is not $list(int)$-instantiated because $list(int)$ is not a constituent of int) and $\alpha_{int}(\{\theta\})$ has models $\big\{ \emptyset, \{y\}, \{z\}, \{x,y,z\} \big\}$. These sets of models correspond respectively to the formulae $x \leftrightarrow z$ and $x \leftrightarrow (y \wedge z)$.

In a classic groundness analysis, the unification $A = [X|Xs]$ is abstracted as $a \leftrightarrow (x \wedge xs)$. Indeed, we assume that any subterm of the term that A is bound to at runtime could unify with any subterm of the terms bound to X or Xs. In the presence of types we know that A and $[X|Xs]$ are both of the same type (otherwise the program is not well-typed). In addition, we know that all unifications between subterms of (the terms bound to) A and $[X|Xs]$ are between terms corresponding to the same types. So in this example (assuming both terms to be of type $list(int)$), we can specify $a \leftrightarrow xs$ for type $list(int)$ and $a \leftrightarrow (x \wedge xs)$ for type int. It is important to note that the interpretations of the variables in $a \leftrightarrow xs$ and in $a \leftrightarrow (x \wedge xs)$ are different. The former refers to subterms of type $list(int)$ whereas the latter refers to subterms of type int. These intuitions are formalised in the following definitions and theorems.

Definition 5. *The τ abstraction of a normalised typed logic program P is obtained by abstracting the equations in P. An equation of the form $X = Y$ with X and Y of type τ' is abstracted by `iff(X, [Y])` (implementing $x \leftrightarrow y$) if $\tau \preceq \tau'$ and otherwise it is abstracted by `true`. An equation of the form $X = f(Y_1, \ldots, Y_n)$ with typed variables $X{:}\tau', Y_1{:}\tau_1, \ldots, Y_n{:}\tau_n$ is abstracted by `iff(X, [`W_1`,...,`W_k`])` (implementing $x \leftrightarrow w_1 \wedge \cdots w_k$) if $\tau \preceq \tau'$ and otherwise it is abstracted by `true` where $\{W_1, \ldots, W_k\} = \{Y_i | 1 \le i \le n,\ \tau \preceq \tau_i\}$.*

Theorem 1. *Correctness. Let E be of the form $X = Y$ or $X = f(Y_1, \ldots, Y_n)$, φ a positive Boolean function on a set of variables V including the variables in E of a type having τ as constituent, and φ' the τ-abstraction of E. If $\theta \in \gamma_\tau(\varphi)$ and $\sigma = mgu(E)$ then $\theta\sigma|_V \in \gamma_\tau(\varphi \wedge \varphi')$.*

Proof. If $X \notin \mathit{Constituents}(\tau')$ then $\varphi' = \mathit{true}$ hence $\varphi \wedge \varphi' = \varphi$. Moreover, $\gamma_\tau(\varphi)$ is closed under instantiation, hence $\theta\sigma|_V \in \gamma_\tau(\varphi \wedge \varphi')$.
Otherwise, observe that $\gamma_\tau(\varphi \wedge \varphi') = \gamma_\tau(\varphi) \cap \gamma_\tau(\varphi')$ hence it suffices to show that $\theta\sigma|_V \in \gamma_\tau(\varphi)$ and $\theta\sigma|_V \in \gamma_\tau(\varphi')$. The former holds because the set $\gamma_\tau(\varphi)$ is closed under instantiation. For the latter, we distinguish:

- Case $X = Y$: $X\theta\sigma = Y\theta\sigma$ hence $\mu_\tau(X\theta\sigma)$ iff $\mu_\tau(Y\theta\sigma)$ and $\theta\sigma|_V \in \gamma_\tau(x \leftrightarrow y) = \gamma_\tau(\varphi')$.
- Case $X = f(Y_1, \ldots, Y_n)$: σ is an mgu hence $\mu_\tau(X\theta\sigma)$ iff $\mu_\tau(f(Y_1, \ldots, Y_n)\theta\sigma)$. $X\theta\sigma$ is definitely instantiated, hence, whether or not $\tau = \tau'$, we have that $\mu_\tau(X\theta\sigma)$ iff $\wedge\{\mu_\tau(Y_i\theta\sigma) | \tau \preceq \tau_i\}$ and $\theta\sigma|_V \in \gamma_\tau(x \leftrightarrow \wedge\{y_i | \tau \preceq \tau_i\}) = \gamma_\tau(\varphi')$.

□

Having shown that the τ-abstraction of unification is correct, we can rely on the results of [7] for the abstraction of the complete program, for the computation of its least model and for the claim that correct answers to a query $\leftarrow p(X_1, \ldots, X_n)$ belong to the concretisation of the $\mathit{Pos}(\mathcal{T})$-formula of the predicate p/n. Basic intuitions are that the $\mathit{Pos}(\mathcal{T})$ formula of a single clause consists of the conjunction of the $\mathit{Pos}(\mathcal{T})$ formulas of the body atoms and that the $\mathit{Pos}(\mathcal{T})$ formula of a predicate consists of the disjunction (lub) of the $\mathit{Pos}(\mathcal{T})$ formulas of the individual clauses defining the predicate.

The adaptation of the implementation technique of Section 2 for an analysis of `append(list(int),list(int),list(int))` is as follows:

Abstraction for type constituent list(int)	*Abstraction for type constituent int*

`append_list_int(Xs,Ys,Zs) :-`
`  iff(Xs,[]),`
`  iff(Ys,[Zs]).`
	`append_int(Xs,Ys,Zs) :-`
	`  iff(Xs,[]),`
	`  iff(Ys,[Zs]).`

`append_list_int(Xs,Ys,Zs) :-`
`  iff(Xs,[Xs1]),`
`  iff(Zs,[Zs1]),`
`  append_list_int(Xs1,Ys,Zs1).`
	`append_int(Xs,Ys,Zs) :-`
	`  iff(Xs,[X,Xs1]),`
	`  iff(Zs,[X,Zs1]),`
	`  append_int(Xs1,Ys,Zs1).`

The least model of `append_int/3` expresses the $\mathit{Pos}(\mathit{int})$-formula $z \leftrightarrow x \wedge y$, i.e. that all subterms of Z of the type *int* are instantiated iff those of X and Y are. The least model of `append_list_int/3` expresses the $\mathit{Pos}(\mathit{list}(\mathit{int}))$-formula $x \wedge (y \leftrightarrow z)$, i.e. that all subterms of X of type $\mathit{list}(\mathit{int})$ are instantiated (in other words the backbone of the list is instantiated when `append/3` succeeds) and that those of Y are instantiated iff those of Z are instantiated. Classical groundness, is obtained by composing the two models, using Equation (1) in Section 3:

```
append(X,Y,Z) :- append_list_int(Xl,Yl,Zl), append_int(Xe,Ye,Ze),
                 iff(X,[Xl,Xe]), iff(Y,[Yl,Ye]), iff(Z,[Zl,Ze]).
```

As $\mathit{int} \preceq \mathit{list}(\mathit{int})$, the analyses of the two constituents is related. In fact, the model of `append_list_int/3` can be used to initialise the fixpoint computation of the model of `append_int/3` as follows from the next proposition (see Appendix A.2 for the proof).

Proposition 2. *Let p/n be a predicate in a typed logic program and τ and τ' types such that $\tau' \preceq \tau$. Let $\bar{Y}$ denote the arguments $Y_i{:}\tau_i$ of p/n such that $\tau' \preceq \tau_i$ but $\tau \not\preceq \tau_i$. Denote the results of the τ- and τ'- analyses of p/n by φ and φ' and let ψ be the conjunction of the variables in $\bar{Y}$. Then $\varphi \wedge \psi \rightarrow \varphi'$.*

5 Polymorphism in *Pos*($\mathcal{T}$)

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that is an instance of the defined type. For example, the `append/3` predicate from Section 4 can be defined with respect to the polymorphic type definition `append(list(T),list(T),list(T))`, stating that its arguments are lists of the same type T. Abstracting `append/3` for this definition results in the same abstractions as in Section 4 but with $list(T)$ and T replacing $list(int)$ and int.

The results of the analysis with type variables is correct for any choice of types instantiating them (e.g., $list(int)$ and int, or $list(list(int))$ and $list(int)$ when `append/3` is called with actual types $list(int)$ or $list(list(int))$ respectively). It is also possible to obtain the result of the analysis for each type-instance by which it is called and sometimes this gives a more precise result. However, it is more efficient to analyze the definition once for its given polymorphic types, and then derive the abstractions of a particular call from that result.

The need for such an approach is even more crucial when analyzing large programs distributed over many modules. It is preferable that an analysis does not need the actual code of the predicates it imports (and of the predicates called directly or indirectly by the imported predicates) but only the result of the call independent analysis. See [25,4,29] for discussions about module based analysis.

Example 4. Consider a predicate `p/2` with both arguments of type $list(list(int))$. The program and its abstractions for the int, $list(int)$ and $list(list(int))$ are as follows:

Concrete definition	*Abstractions*

`p(X,Y):- append(X,X,Y).`	`p_list_list_int(X,Y) :- append_list_T(X,X,Y).`
	`p_list_int(X,Y) :- append_T(X,X,Y).`
	`p_int(X,Y) :- append_T(X,X,Y).`

Intuitively, it is clear that the constituent $list(list(int))$ from the call to `append/3` corresponds to the constituent $list(T)$ in `append/3`'s definition. Hence, instead of computing the $list(list(int))$-abstraction of `append/3`, its $list(T)$-abstraction can be used. Likewise, the constituents $list(int)$ and int from the call both correspond to the constituent T in the definition, hence also their abstractions.

To formalize the relationship between the constituents of the actual types and those of the formal types, we introduce the notion of a *variable typing.*

Definition 6. *A* variable typing *ζ is a mapping from program variables to types; $dom(\zeta)$ is the set of variables for which the mapping is defined; $X\zeta$ denotes the type of X under the variable typing ζ. Given variable typings ζ and ζ' over the* *same domain, ζ' is an instance of ζ if , for every $X \in dom(\zeta)$ there exists a type substitution ξ such that $X\zeta' = X\zeta\xi$.*

The notion of *Constituents* extends in a straightforward way to variable typings. In a well-typed program, the types of the arguments of a call are always at least as instantiated as the types of the corresponding arguments in the predicate definition. Hence the constituents occurring in a variable typing ζ' of a call $p(X_1, \ldots, X_n)$ can be related to the constituents of the variable typing ζ of the head $p(Y_1, \ldots, Y_n)$ as follows:

Definition 7. *Consider variable typings ζ' and ζ with ζ' an instance of ζ. The* type mapping *between ζ and ζ' is the minimal relation $R^{\zeta}_{\zeta'} : \mathcal{T} \times \mathcal{T}$ such that:*

- *If $X/\tau_1 \in \zeta'$ and $X/\tau_2 \in \zeta$ then $(\tau_1, \tau_2) \in R^{\zeta}_{\zeta'}$.*
- *If $(\tau_1, \tau_2) \in R^{\zeta}_{\zeta'}$ and $\tau_2 \in V_{\mathcal{T}}$ then $\forall \tau \in Constituents(\tau_1) : (\tau, \tau_2) \in R^{\zeta}_{\zeta'}$.*
- *If $(\tau_1, \tau_2) \in R^{\zeta}_{\zeta'}$ and $\tau_2 = h(V_1, \ldots, V_n)\xi_2$ and $\tau_1 = h(V_1, \ldots, V_n)\xi_1$ and $h(V_1, \ldots, V_n) \longrightarrow f_1(\tau_{1_1}, \ldots, \tau_{1_{m_1}})\,; \ldots\,;\, f_l(\tau_{l_1}, \ldots, \tau_{l_{m_l}})$ is the type rule for h/n then, for all τ_{i_j}, $(\tau_{i_j}\xi_1, \tau_{i_j}\xi_2) \in R^{\zeta}_{\zeta'}$.*

The type function $\phi^{\zeta}_{\zeta'} : \mathcal{T} \mapsto 2^{\mathcal{T}}$ *is defined for constituents of the most instantiated type ζ': $\phi^{\zeta}_{\zeta'}(\tau) = \{\tau' | (\tau, \tau') \in R^{\zeta}_{\zeta'}\}$.*

When ζ and ζ' are obvious from the context, we will write R and $\phi(\tau)$.

Example 5. The type mapping and type function associated to the variable typings $\zeta = \{X/list(T)\}$ and $\zeta' = \{X/list(list(int))\}$ are repectively:

- $R = \{(list(list(int)), list(T)),\ (list(int), T),\ (int, T)\}$ and
- $\phi = \{(list(list(int)), \{list(T)\}),\ (list(int), \{T\}),\ (int, \{T\})\}$.

If ζ' denotes the variable typing associated to the variables of the call, and ζ denotes the variable typing associated to the called predicate's head variables, the mapping R expresses that the τ-abstraction of a call corresponds to the $\phi(\tau)$-abstraction of the called predicate. As an illustration, ϕ from Example 5 gives the mappings used in Example 4.

However, in general $\phi(\tau)$ is not a singleton, as a constituent of the call can be mapped to itself *and* to one or more type variables in the polymorphic type definition. Consider for example a predicate `q/4` and its abstractions:

Concrete	*Abstraction w.r.t. int*	*Abstraction w.r.t.* `T`

`pred q(int,int,T,T).`
`q(X,Y,U,V):- X=Y,U=V.`
`q(X,Y,U,V):- X=0.`
	`q_int(X,Y,U,V):-`
	`  iff(X,[Y]).`
	`q_int(X,Y,U,V):-`
	`  iff(X,[]).`
		`q_T(X,Y,U,V):-`
		`  iff(U,[V]).`
		`q_T(X,Y,U,V).`

Suppose `q(A,B,C,D)` is a call with A, B, C and D of type *int*. The type substitution on the defined types is $\{T/int\}$, and hence $\phi(int) = \{int, T\}$. A correct *int*-abstraction of the *call* should include both the *int*-abstraction and the T-abstraction from the definition, since in the polymorphic analysis, it was assumed that T does not have *int* as constituent. Hence, the *int*-abstraction of the call becomes $q\_int(A, B, C, D) \wedge q\_T(A, B, C, D)$. The formal definition of the τ-abstraction of a predicate call is then as follows:

Definition 8. *Let ζ' and ζ be the variable typings associated respectively to a call $p(X_1, \ldots, X_n)$ and the head of p's definition; let $\tau' \in Constituents(\zeta')$; The τ'-abstraction of $p(X_1, \ldots, X_n)$ is defined as $\wedge\{p\_\tau(X_1, \ldots, X_n) | \tau \in \phi(\tau')\}$.*

Handling a predicate call in such a way, however, possibly causes some precision loss. Indeed, one can verify that a *monomorphic* analysis of `q(X,Y,U,V)` for the type `p(int,int,int,int)` gives the formula $((x \leftrightarrow y) \wedge (u \leftrightarrow v)) \vee x$ while the conjunction of calls $p\_int(X, Y, U, V), p\_T(X, Y, U, V)$ gives the less precise formula $((x \leftrightarrow y) \vee x) \wedge ((u \leftrightarrow v) \vee true)$. However, the result is always safe:

Theorem 2. *Let p be a predicate definition and ζ and ζ' variable typings for the variables of p such that ζ' is an instance of ζ. For $\tau \in Constituents(\zeta)$, let φ^τ denote the τ-abstraction of p for the variable typing ζ, and similarly for $\tau' \in Constituents(\zeta')$, let $\psi^{\tau'}$ denote the τ'-abstraction of p for the variable typing ζ'. Then it holds that $\psi^{\tau'} \to \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.*

Theorem 2 states that the use of Definition 8 safely approximates the result obtained by an analysis of the called predicate for the type instances used in the call. As the above example illustrates, the equivalence does not hold and some precision loss is possible. However, such precision loss occurs only when constituents of parametric type instances interfere with each other or with other types. In case there is no such interference (i.e. $\phi(\tau')$ is a singleton), the results of call-independent analysis equal those of call-dependent analysis. We believe that interference is more the exception than the rule. Indeed, it only happens when type instances of two different type variables have a common constituent or when the instance of a type variable has a constituent common with the type of another argument. Even if the types do interfere, a loss of precision is avoided if the lub of the *Pos*($\mathcal{T}$)-formulas of each individual clause body is equal to the *Pos*($\mathcal{T}$)-formula of one of the clause bodies (the same for all types τ). This could well be often the case in practice because different clauses of a program tend to establish the same degree of instantiatedness (or in case of recursion, the *Pos*($\mathcal{T}$)-formula of the base case implies the *Pos*($\mathcal{T}$)-formula of the recursive clause(s)):

Corollary 1. *With the same notations as in Theorem 2, let φ_j^τ denote the τ-abstraction of the body of clause j of predicate p's definition. If $\phi(\tau')$ has only one element or there exists a clause k such that for all $\tau \in \phi(\tau')$: $\varphi_k^\tau = \vee\{\varphi_j^\tau | j \text{ is a clause}\}$, i.e. the τ-abstraction of the body of clause k is the upper-bound of the τ-abstractions of all clause bodies, then $\psi^{\tau'} = \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.*

Intuitively, one can say that some dependencies are ignored by analysing the predicate separately for each element in $\phi(\tau')$, and that they are only partly recovered by taking the conjunction of the obtained formulas. We refer to Appendix A.1 for the proofs of Theorem 2 and Corollary 1.

6 Discussion

The main goal of this research is to improve the precision of groundness dependencies by taking into consideration type information. Precision is improved for two reasons: (a) when unifying terms (or variables) of the same type it is possible to refine the resulting dependencies (e.g. list elements with list elements, and list backbone with list backbone); and (b) computation paths which unify terms of different types can be pruned.

While $Pos(\mathcal{T})$ can improve the precision of groundness dependencies it is important also to note that the $Pos(\mathcal{T})$ formulas provide valuable information on their own. In future work we intend to use it to improve automated termination analysis which in turn will be used in a binding-time analysis for the off-line specialisation of logic programs (improving the work in [3].).

Some preliminary experiments, using the implementation technique of [7] indicate that our approach can be done more efficiently than the approach in [19] where a single model is computed, and where a program variable corresponds to several Boolean variables, one for each type constituent. The reason is that the computation of the least fixpoint of the abstracted program slows down more than linearly with the number of variables. Moreover, proposition 2 allows for extra savings by reusing results from one type constituent in the computation for other constituents.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Efficient structural information analysis for real CLP languages. In *Proc. LPAR2000*, volume 1955 of *LNCS*. Springer, 2000.
3. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In *Proc. ESOP'98*, volume 1381 of *LNCS*, pages 27–41. Springer, 1998.
4. F. Bueno, M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In *Preproceedings LOPSTR2000*, 2000.
5. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Logic Programming*, 38(3):354–370, 1999.
6. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. SAS'94*, volume 864 of *LNCS*, pages 281–296. Springer, 1994.
7. M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, 1995.
8. M. Codish, K .Marriott, and C. Taboch. Improving program analyses by structure untupling,. *Journal Logic Programming*, 43(3), June 2000.
9. A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: propositional formula as abstract domain for groundness analysis. In *Proc. LICS'91*, pages 322–327. IEEE Press, 1991.
10. A. Cortesi, G. Filé, and W. H. Winsborough. Optimal groundness analysis using propositional logic. *J. Logic Programming*, 27(2):137–167, 1996.

11. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, 2000.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'97*, pages 238–252, 1977.
13. B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In *Proc. CP'99*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.
14. J. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pages 351–365. MIT Press, 1995.
15. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In *Proc. ICLP'94*, pages 599–613. MIT Press, 1994.
16. M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Debugging, and optimization using the Ciao system preprocessor. In *Proc. ICLP'99*, pages 52–66, 1999.
17. P. Hill and J. Lloyd. *The Gödel Language.* MIT Press, 1994.
18. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic programming*, 13(2&3):205–258, 1992.
19. V. Lagoon and P. J. Stuckey. A framework for analysis of typed logic programs. In *Proc. FLOPS 2001*, volume 2024 of *LNCS*, pages 296–310. Springer, 2001.
20. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.
21. G. Levi and F. Spoto. An experiment in domain refinement: Type domains and type representations for logic programs. In *Proc. PLILP/ALP'98*, volume 1490 of *LNCS*, pages 152–169. Springer, 1998.
22. J.W. Lloyd. *Foundation of Logic Programming.* Springer, second edition, 1987.
23. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.
24. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *Proc. ICLP'95*, pages 781–795. MIT Press, 1995.
25. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In *Proc. ICLP Workshop on Optimization and Implementation of Declarative Languages*, 1999.
26. O. Ridoux, P. Boizumault, and F. Malésieux. Typed static analysis: Application to groundness analysis of Prolog and lambda-Prolog. In *Proc. FLOPS'99*, volume 1722 of *LNCS*, pages 267–283. Springer, 1999.
27. J.G. Smaus, P. M. Hill, and A. King. Mode analysis domains for typed logic programs. In *Proc. LOPSTR'99*, volume 1817 of *LNCS*, pages 82–101. Springer, 2000.
28. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1–3):17–64, October–November 1996.
29. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In *Proc. LPAR2000*, volume 1955 of *LNCS*, pages 399 – 416, Springer, 2000.
30. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In *Proc. ICLP'99*, pages 500 – 514. MIT Press, 1999.

A Proofs

A.1 Proof of Theorem 2 and Corollary 1

In the formulation and proofs that follow, we consider a program construct under two variable typings ζ and ζ' such that ζ' is an instance of ζ. With X a program variable, we denote with τ_X and τ'_X the type of X in respectively ζ and ζ'. We relate the τ'-abstraction of the construct for a type $\tau' \in \mathit{Constituents}(\zeta')$ with its $\phi(\tau')$-abstractions. First, we consider a single unification.

Lemma 1. *With E an equality, let φ^{τ} denote its τ-abstraction for the variable typing ζ and and $\psi^{\tau'}$ its τ'-abstraction for the variable typing ζ'. Then it holds that $\psi^{\tau'} = \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.*

Proof. E is of the form $X = \ldots$. Assume $Y_1, \ldots Y_n$ are the variables in the rhs. such that $\tau' \in \mathit{Constituents}(\tau'_{Y_i})$. Then $\psi^{\tau'} = x \leftrightarrow y_1 \wedge \ldots \wedge y_n$. Let $\phi(\tau') = \{\tau_1, \ldots, \tau_n\}$. The variables $Y_{i_1}, \ldots, Y_{i_{n_i}}$ such that $\tau_i \in \mathit{Constituents}(\tau_{Y_{i_j}})$ are a subset of $\{Y_1, \ldots, Y_n\}$ and $\varphi^{\tau_i} = x \leftrightarrow y_{i_1} \wedge \ldots \wedge y_{i_{n_i}}$. Moreover, by construction of ϕ it holds that $\tau' \in \mathit{Constituents}(\tau'_{Y_i})$ if and only if $\exists\, \tau_j \in \mathit{Constituents}(\tau_{Y_i})$ and hence $\bigcup_i \{Y_{i_1}, \ldots, Y_{i_{n_i}}\} = \{Y_1, \ldots, Y_n\}$. Consequently,

$$\begin{aligned}\psi^{\tau'} &= x \leftrightarrow y_1 \wedge \ldots \wedge y_n \\ &= (x \leftrightarrow y_{1_1} \wedge \ldots \wedge y_{1_{n_1}}) \wedge \ldots \wedge (x \leftrightarrow y_{k_1} \wedge \ldots \wedge y_{k_{n_k}}) \\ &= \varphi^{\tau_1} \wedge \ldots \wedge \varphi^{\tau_k}\end{aligned}$$
□

Some properties of propositional logic (b is a propositional formula):

$$\text{If } \forall i : b'_i \rightarrow b_i \text{ then } \bigwedge_i b'_i \rightarrow \bigwedge_i b_i \tag{1}$$

$$\text{If } \forall i : b'_i \rightarrow b_i \text{ then } \bigvee_i b'_i \rightarrow \bigvee_i b_i \tag{2}$$

$$\forall i, j : \bigvee_j (\bigwedge_i b_{i,j}) \rightarrow \bigwedge_i (\bigvee_j b_{i,j}) \tag{3}$$

The following lemma concerns the τ'- and τ-abstractions of a predicate definition.

Lemma 2. *Let p be a predicate definition; let φ^{τ} and $\varphi^{\tau}_{i,j}$ denote respectively the τ-abstraction of p and of $B_{i,j}$, the ith body atom in the jth clause, for the type assignment ζ, and similarly, $\psi^{\tau'}$ and $\psi^{\tau'}_{i,j}$ the τ'-abstraction of p and $B_{i,j}$ for the variable typing ζ'. If for each body atom $B_{i,j}$: $\psi^{\tau'}_{i,j} \rightarrow \wedge\{\varphi^{\tau}_{i,j} | \tau \in \phi(\tau')\}$ then $\psi^{\tau'} \rightarrow \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.*

Proof. Assuming $\psi^{\tau'}_{i,j} \rightarrow \wedge\{\varphi^{\tau}_{i,j} | \tau \in \phi(\tau')\}$, we prove $\psi^{\tau'} \rightarrow \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.
Let $B_{1,j}, \ldots, B_{n_j,j}$ be the body of the jth clause; let $\varphi^{\tau}_j = \wedge\{\varphi^{\tau}_{i,j} | i \in \{1, \ldots, n_j\}\}$ and $\psi^{\tau'}_j = \wedge\{\psi^{\tau}_{i,j} | i \in \{1, \ldots, n_j\}\}$ be their respectively τ- and τ'-abstraction. Using the assumption, (1) and commutativity of $\wedge$, we obtain: $\psi^{\tau'}_j \rightarrow \wedge\{\varphi^{\tau}_j | \tau \in \phi(\tau')\}$.
Using (2) we obtain: $\bigvee_j \psi^{\tau'}_j \rightarrow \bigvee_j (\bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}_j)$.
Using (3) we obtain: $\bigvee_j \psi^{\tau'}_j = \psi^{\tau'} \rightarrow \bigwedge_{\tau \in \phi(\tau')} (\bigvee_j \varphi^{\tau}_j = \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau})$. □

From the above lemma, we derive two corollaries that we can use to prove Corollary 1 (p. 415). A first one handles the first option in the condition of Corollary 1, namely the case that $\phi(\tau') = \{\tau\}$, that is, when there is a one-to-one correspondence between τ' in ζ' and τ in ζ.

Corollary 2. *With the same notations as in Lemma 2, if $\phi(\tau')$ has only one element and for each body atom: $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^{\tau} | \tau \in \phi(\tau')\}$ then $\psi^{\tau'} = \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.*

Proof. Immediate from the proof of Lemma 2. □

The second corollary handles the case that the least upperbound of the τ-abstractions of the individual clauses equals the τ-abstraction of one of the clauses.

Corollary 3. *With the same notations as in Lemma 2, if $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^{\tau} | \tau \in \phi(\tau')\}$ and there exists a clause k such that for all $\tau \in \phi(\tau')$:
$\varphi_k^{\tau} = \vee\{\varphi_j^{\tau} | j \text{ is a clause}\}$ then $\psi^{\tau'} = \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.*

Proof. With the notations of in Lemma 2 we now obtain $\psi_j^{\tau'} = \wedge\{\varphi_j^{\tau} | \tau \in \phi(\tau')\}$.
Hence $\bigvee_j \psi_j^{\tau'} = \psi^{\tau'} = \bigvee_j \big(\bigwedge_{\tau\in\phi(\tau')} \varphi_j^{\tau}\big)$.
Using the assumption, $\bigvee_j \big(\bigwedge_{\tau\in\phi(\tau')} \varphi_j^{\tau}\big) = \bigwedge_{\tau\in\phi(\tau')} \varphi_k^{\tau} = \bigwedge_{\tau\in\phi(\tau')} \varphi^{\tau}$. □

Now, we can prove Theorem 2 by using Lemmas 1 and 2 from above.

Theorem 2. *Let p be a predicate definition; let φ^{τ} denote the τ-abstraction of p for the variable typing ζ; let $\psi^{\tau'}$ denote the τ'-abstraction of p for the variable typing ζ'. Then it holds that $\psi^{\tau'} \to \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.*

Proof. The formula $\psi^{\tau'}$ is computed by a bottom up fixpoint operator that, starting from the formula $false$ for all predicates, recomputes the $Pos(\mathcal{T})$-formula of each predicate until a fixpoint is reached. The proof is by induction on the number of iterations.

The predicate p is defined by the clauses $C_1, \ldots, C_m$, and a clause C_i is a conjunction of body atoms $B_{i,1}, \ldots, B_{i,n_i}$.

– First iteration. Since Lemma 1 holds for the unifications, and the predicate calls in the body only introduce the formula $false$, we have for each body atom $B_{i,j}$ that $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^{\tau} | \tau \in \phi(\tau')\}$ where $\psi_{i,j}^{\tau'}$ denotes the τ'-abstraction of $B_{i,j}$ for variable typing ζ' and $\varphi_{i,j}^{\tau}$ denotes the τ-abstraction of $B_{i,j}$ for variable typing ζ. Applying Lemma 2 results in $\psi^{\tau'} \to \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$.
– Induction step. Assume the formula holds after k iterations. For the abstraction of p as constructed in iteration $k + 1$, we have either by Lemma 1 (for the unifications) or by the induction hypothesis (for the predicate calls) that for each body atom $B_{i,j}$ holds that $\psi_i^{\tau'} \to \wedge\{\varphi_{i,j}^{\tau} | \tau \in \phi(\tau')\}$.
  Applying Lemma 2 on obtains: $\psi^{\tau'} \to \wedge\{\varphi^{\tau} | \tau \in \phi(\tau')\}$. □

Corollary 1 on page 415 can be proven analogously to the proof of Theorem 2, using Corollaries 2 and 3 instead of Lemma 2.

A.2 Proof of Proposition 2

Proposition 2. *Let p/n be a predicate in a typed logic program and τ and τ' types such that $\tau' \preceq \tau$. Let $\bar{Y}$ denote the arguments $Y_i{:}\tau_i$ of p/n such that $\tau' \preceq \tau_i$ but $\tau \not\preceq \tau_i$. Denote the results of the τ- and τ'- analyses of p/n by φ and φ' and let ψ be the conjunction of the variables in $\bar{Y}$. Then $\varphi \wedge \psi \rightarrow \varphi'$.*

Proof. Let us denote by $\bar{X}$ the arguments of p/n not in $\bar{Y}$ and write $\varphi(\bar{X}, \bar{Y})$, $\varphi'(\bar{X}, \bar{Y})$ and $\psi(\bar{Y})$ instead of φ, φ' and ψ. Moreover we assume without loss of generality that the arguments of p/n are $p(\bar{X}, \bar{Y})$.

The proof is based on the observation is that there is a one-one correspondence between value assignments $\{\overline{x}/\overline{v}\}$ that are models of the τ-abstraction $\varphi(\overline{x})$ of a predicate q/m and the atoms $q(\overline{v})$ that are computed during the computation of the least fixpoint of the abstracted program. With P_τ and $P_{\tau'}$ respectively the τ- and τ'-abstractions of the program, we proof by induction that $p(\bar{u}, \bar{v}) \in T_{p_\tau} \uparrow k$ implies $p(\bar{u}, \overline{true}) \in T_{p_{\tau'}} \uparrow k$. (We assume the least fixpoint of the `iff/2` predicate is computed in advance.)

Iteration 1. If $p(\bar{u}, \bar{v}) \in T_{P_\tau} \uparrow 1$, it means that P_τ contains a clause $c_\tau = p(\bar{X}, \bar{Y}) \leftarrow B_1, \ldots, B_m$ with all B_i calls to `iff/2` (τ-abstractions of equalities) and that there is a substitution $\theta = \{\bar{X}/\bar{u}, \bar{Y}/\bar{v}, \bar{Z}_1/\bar{w}_1\}$ such that $\forall i : B_i\theta$ is true in the least model of `iff/2` (note that the variables of $\bar{Y}$ do not occur in the body). Hence $P_{\tau'}$ contains a clause $c_{\tau'} = p(\bar{X}, \bar{Y}) \leftarrow B'_1, \ldots, B'_m, C_1, \ldots, C_n$ with the B'_i τ'-abstractions of the equalities having B_i as their τ-abstraction and the C_i τ'-abstractions of equalities having *true* as τ-abstraction. To prove that $p(\bar{u}, \overline{true}) \in T_{P_{\tau'}} \uparrow 1$ it suffices to prove that $\sigma = \{\bar{X}/\bar{u}, \bar{Y}/\overline{true}, \bar{Z}_1/\bar{w}_1, \bar{Z}_2/\overline{true}\}$ makes the body of $c_{\tau'}$ *true* (the variables of $\bar{Z}_2$ occur in the body of $c_{\tau'}$ but not in the body of c_τ). To prove the latter, we show that the three kinds of atoms in the body of $c_{\tau'}$ are *true* in the least model of `iff/2` under the substitution σ:

– B_i is the τ-abstraction of an equation $U = V$. Obviously, $B'_i = B_i$, and $B_i\sigma = B_i\theta$ is true in the least model of `iff/2`.

– B_i is the τ-abstraction of an equation $U = f(\bar{V})$. B_i is of the form $\mathit{iff}(U, [\overline{V_\tau}])$ with $\overline{V_\tau}$ the variables of $\overline{V}$ having τ as a constituent of their type. Hence B'_i is of the form $\mathit{iff}(U, [\overline{V_\tau}, \overline{V_{\tau'}}])$ with $\overline{V_{\tau'}}$ the variables of $\overline{V}$ having τ' but not τ as a constituent of their type. Note that σ binds the variables of $\overline{V_{\tau'}}$ to *true*. We know that $B_i\theta$ is *true*, that means either U and $\overline{V_\tau}$ are bound to *true* and thus also $B_i\sigma$ is *true* or U and at least one of $\overline{V_\tau}$ are bound to $false$, also in this case, $B_i\sigma$ is *true*.

– C_i is of the form $\mathit{iff}(\ldots)$ with all variables not occurring in the body of c_τ. All these variables are bound to *true* under σ hence $C_i\sigma$ is true.

Iteration $k+1$. Assume $p(\bar{u}, \bar{v}) \in T_{P_\tau} \uparrow k + 1$. The difference with the first iteration is the form of the clauses c_τ and $c_{\tau'}$. The body of c_τ now contains also calls of the form $q(\bar{X}', \bar{Y}')$ (with $Y'_i : \tau_i$ the arguments of q/m such that $\tau' \preceq \tau_i$ but $\tau \not\preceq \tau_i$). The body of $c_{\tau'}$ contains corresponding calls $q(\bar{X}', \bar{Y}')$. We know that $q(\bar{X}', \bar{Y}')\theta \in T_{P_\tau} \uparrow k$, hence, by induction hypothesis, $q(\bar{X}', \bar{Y}')\sigma \in T_{P_{\tau'}} \uparrow k$. The other atoms in the body of $c_{\tau'}$ are true for the same reasons as in the first iteration, hence the whole body is true under σ and $p(\bar{u}, \overline{true}) \in T_{P_{\tau'}} \uparrow k + 1$.

A Prolog Tailoring Technique on an Epilog Tailored Procedure

Yoon-Chan Jhi[1], Ki-Chang Kim[1], Kemal Ebcioglu[2], and Yong Surk Lee[3]

[1]Dept. of Computer Science & Engineering, Inha University
kchang@inha.ac.kr
[2]IBM T.J. Watson Research Center, Yorktown Heights
[3]Dept. of Electrical and Electronic Engineering, Yonsei University

Abstract. Prolog tailoring technique, an optimization method to improve the execution speed of a procedure, is proposed in this paper. When a procedure is repeatedly called and the machine has a lot of callee-saved registers, optimizing prolog and epilog of the procedure can become an important step of optimization. Epilog tailoring supported by IBM xlc compiler has been known to improve a procedure's execution speed by reducing the number of register-restoring instructions on exit points. In this paper, we propose a prolog tailoring technique that can reduce register-saving instructions at entry points. We can optimize prolog by providing multiple tailored versions of it on different execution paths of the procedure and by delaying the generation of register-saving instructions as late as possible on each path. However, generating prolog inside diamond structures or loop structures will cause incorrectness or unnecessary code repetition. We propose a technique to generate efficient prolog without such problems based on Tarjan's algorithms to detect SCCs (Strongly Connected Components) and BCCs (Bi-Connected Components).

1. Introduction

In a procedure call, some registers, called callee-saved registers, should preserve their values across the call; that is their values should be the same before and after the call. The callee guarantees it by saving those registers before starting the actual function body and restoring them later before leaving the code [2,3]. The register-saving instructions are called a prolog, while the register-restoring ones called an epilog. Every time this procedure is called, the prolog and epilog should be executed. For frequently executed procedures, therefore, they consume significant amount of time and are an important source of optimization [4,5].

In order to reduce the overhead of prolog and epilog code, the traditional technique was to compute those callee-saved registers that are actually killed inside the procedure. They are, then, saved and later restored in the prolog and epilog code, respectively. In this paper, we propose techniques to further reduce the number of registers that need to be saved and restored by tailoring the prolog and epilog to different execution paths. We observe that if the procedure has several execution paths, and if each path is modifying different sets of callee-saved registers, then, we may provide a different pair of prolog and epilog for each path. Since they are saving and restoring only those registers that are killed in the particular path, we can reduce the size of them.

Tailoring epilog has been implemented in some compilers, e.g. IBM xlc compiler, and the algorithm is explained in [12]. In [5], a brief mention on prolog tailoring has been made, but detailed algorithm is not published yet. In this paper, we provide the detailed algorithm and examples of prolog tailoring. The paper is organized as follows. Section 2 explains the existing epilog tailoring technique and some related research. Section 3 explains the basic idea of the proposed prolog tailoring technique. Section 4 describes the proposed prolog tailoring algorithm in detail. Section 5 and 6 gives out experimental results and a conclusion. Finally, Appendix A will show an example.

2. Epilog Tailoring and Related Researches

Epilog tailoring tries to minimize the number of register-restoring operations at each exit point. The basic technique is to split the exit point. By splitting it, the set of killed registers (therefore the set of should-be-restored registers) can be different at different exit points, and we can restore only those actually killed registers at each exit point.

Fig. 1 shows an example. Fig. 1(a) is the prolog and epilog code generated without any tailoring process. In the procedure, r28, r29, r30, and r31 are killed; therefore they are saved and restored at the entrance and exit points. Fig. 1(b) shows the same procedure with tailored epilog code. The original exit point is split into two: e1 and e2 in the figure. At the paths reaching e1, the first exit point, r28 and r30 are killed, while at the path reaching e2, r29 and r31 are killed. Therefore we can have a different (and a smaller) epilog code at each exit point. The second exit point, e2, may be split into two again, optimizing the epilog code further. The procedure in Fig. 1(a) will execute 4+4 register saving/restoring operations, while that in Fig. 1(b) will execute 4+2 register saving/restoring operations regardless of which exit point it takes.

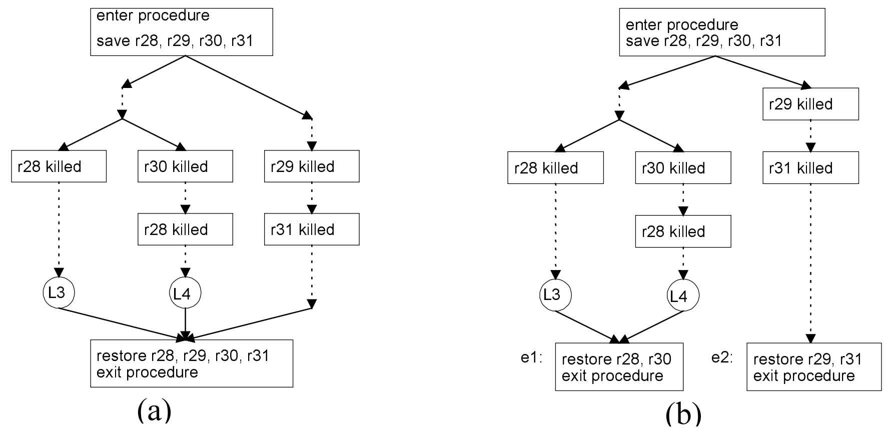


Fig. 1. Applying epilog tailoring technique on a procedure

Other efforts to optimize prolog/epilog of procedures have been reported in [5,6,7,10]. In [5], Huang investigates the reuse of output results of some basic blocks during the run time when the same input values to them are detected. Not all basic blocks are reusable, because the input values are rarely identical for different executions of the basic blocks. But a prolog basic block is a good candidate for such reusing technique, because a procedure is often called with the same parameters. In [6,7,10], the output reusing is reduced to a single instruction. Prolog and epilog again provide a good source of instructions for such technique. Both cases do not reduce the absolute number of prolog/epilog instructions as in our case.

3. Prolog Tailoring

The basic idea of prolog tailoring is to push down the location of the register-saving operations along the execution paths as close as possible to the point where the registers are actually killed. Fig. 2 shows how prolog codes are generated on the same code as in Fig. 1(b). It saves only 2 registers at all entrance points, while the code in Fig. 1(b) saves 4. As the result, regardless of which path the procedure takes in the run time, the code in Fig. 1(b) expends 6 operations in register saving/restoring, while the code in Fig. 2 expends 4 operations.

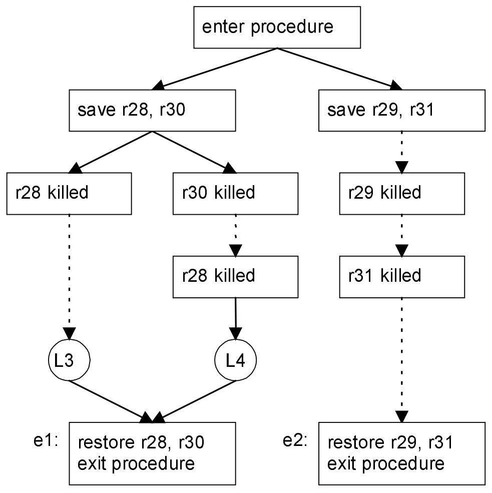


Fig. 2. Applying prolog tailoring technique on an epilog tailored procedure

One important question in prolog tailoring is how far we can push down the register-saving operations. If a basic block is killing register r1, the saving operation for r1 can be pushed down to the entrance point of the basic block. If a register is killed in several basic blocks that have a common parent node, its corresponding saving operation can move down to the entrance point of the basic block where the parent node belongs to. Moving it further down will cause duplication of register-saving operations. If a register is killed inside a loop, the corresponding saving operation should be stopped before the loop. Once entering the loop, the register-saving operation will be executed repeatedly, wasting the CPU time. Finally, if a register is killed inside a diamond structure, e.g. if-then-else structure, the corresponding saving operation should be located before the structure, unless the join point of this diamond is split.

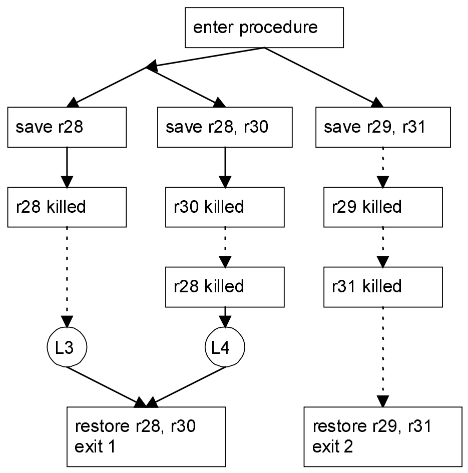


Fig. 3. Register-saving code generated inside a diamond structure

Pushing register-saving operations inside a diamond structure may modify the semantics of the original code. Fig. 3 shows an example. In the figure, we have pushed down the register-saving operations into a diamond structure to make them closer to the destruction points. The path reaching L3 kills only r28, while the path reaching L4 kills r28 and r30. Therefore, the code in Fig. 3 saves r28 on L3 path and r28 and r30 on L4 path. However, at exit 1, the jointing point of L3 and L4 path, r28 and r30 both are restored. If the program took L3 path during the run time, we are saving r28 only and restoring r28 and r30. Since the stack frame does not contain the original value of r30, the final value restored in r30 becomes unpredictable.

In this paper, we propose algorithms to push down register-saving operations as close as possible to the actual destruction points but not with unnecessary duplication nor with incorrect modification of the original program's semantics.

4. Prolog Tailoring Algorithm

We assume a control glow graph is already built for a procedure for which we want to add prolog and epilog. Further, we assume the epilog is already tailored as explained in Section 2. The first step to tailor the prolog is to detect diamond and loop structures and replace them with a single node. With this replacement, the control flow graph will become a tree. On this tree, we compute DKR (Definitely Killed Registers) at each node and determine which register should be saved where, based on these DKRs. The overall algorithm is in Fig. 4, and its major steps are explained in the following sections.

```
basicblockfg_t generate_prolog(basicblockfg_t bbfg)
{
    sccfg ← remove loops from basic block flow graph;
    bccfg ← remove diamond structures from sccfg;
    dkr ← compute DKR(Definitely Killed Register) for each
node in bccfg;
    tailored_bbfg ← generate register-saving operations on
bbfg based on dkr and bccfg;
    return tailored_bbfg ;
}
```

Fig. 4. Basic steps of prolog tailoring

4.1 Removing Loops

The first step of prolog tailoring is to remove loops. Loops can be identified as SCCs (Strongly Connected Components), and we can use Tarjan's algorithm [9] to detect them. Fig. 5 shows how a loop is replaced with a single node in a control flow graph. Node 2, 3, and 4 in Fig. 4(a) form an SCC; they are replaced with a single node as in Fig. 4(b). All edges reaching node 2, 3, and 4 should also reach the new replaced node, and all leaving edges from them should also leave from the new node. The new graph with all loops removed is called an SCC flow graph.

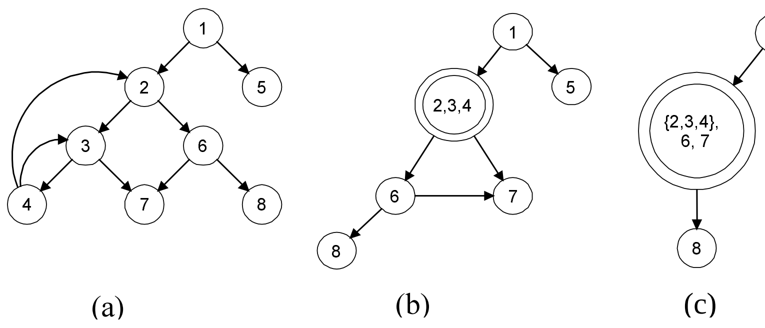


Fig. 5. Constructing SCC flow graph and BCC flow graph

4.2 Removing Diamond Structures

The second step is to remove all diamond structures on the SCC flow graph. The modified graph is called a BCC flow graph. We can detect diamond structures by detecting BCCs (Bi-Conected Components) [1, 9]. To define a BCC, let's define a bi-connected graph and an articulation point as in [1]. An articulation point is a node in a graph that divides the graph into two or more sub-graphs when it is removed. A bi-connected graph is one that does not have any articulation point. A BCC is a bi-connected sub-graph inside a full graph. Node {2,3,4}, 6, and 7 in Fig. 5(b) form a BCC; therefore they form a diamond structure. By replacing them with a single node, we get Fig. 5(c).

The BCCs detected by Tarjan's algorithm may contain shared nodes, nodes contained in more than one BCC. We need a systematic way to decide the membership of such a shared node.

Table 1. BCC set found in Fig. 5(b)

BCC	1	2	3	4
SCC	1, {2, 3, 4}	1, 5	{2, 3, 4}, 6, 7	6, 8

Table 1 shows the four BCCs found in Fig. 5 by Tarjan's algorithm. In the table, node 6 belongs to BCC node 3 and 4; therefore it is a shared node. The algorithm to remove shared nodes is in Fig. 6. In the algorithm, a local root node of a BCC is an entrance node to that BCC. The overall algorithm to obtain a BCC flow graph from an SCC flow graph is in Fig. 7.

```
bccset_t  remove_shared_node(bccset_t bccset)
{
  for(all BCCs in bccset)
    if(its local root nodes has outgoing edges from this BCC)
       remove this local root node;
  for(all BCCs in bccset)
    if(there is a shared node){
       remove the shared node in the parent BCC;
         if(the parent BCC becomes empty)
            remove the parent bcc from bccset;
        }
  if(the root of sccfg was removed){
    generate a BCC that includes the root of sccfg as the
only member;
    add this BCC into bccset;
  }
  return bccset ;
}
```

Fig. 6. Algorithm for shared node removal in a given BCC set

```
bccfg_t  scc_to_bcc(sccfg_t  sccfg)
{
    bccset ← detect all BCCs from sccfg;
    bccset ← remove_shared_node(bccset) ;
    bccfg ← add links among BCCs in bccset based on the edges
in sccfg;
    return  bccfg ;
}
```

Fig. 7. Algorithm for constructing a BCC flow graph from a given SCC flow graph

4.3 Computing DKRs

The third step is to compute killed registers at each node in the BCC flow graph and to compute DKRs based on them. A DKR of a BCC node represents a set of registers that are definitely killed in all paths starting from this BCC node. Fig. 8 shows a BCC flow graph with killed-registers and a DKR at each node. For example, at node 1, we can see r27 is killed inside node 1, and r28 is killed at both paths starting from node 1; therefore the DKR for node 1 includes r27 and r28. The DKR of node n can be defined recursively as follows.

$$DKR(n) = \bigcap_{\text{for } j \in child(n)} DKR(j) + killed\_reg(n)$$

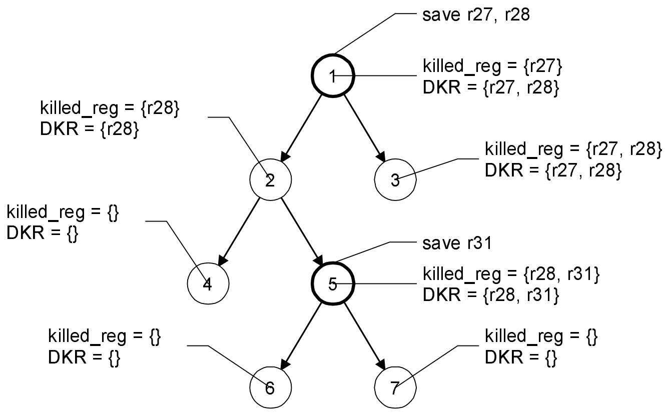


Fig. 8. Computing DKR of each node and generating register-saving instructions

4.4 Prolog Generation

The last step is to generate register-saving operations. We start from the root node in the BCC flow graph moving down. At each node visited, we generate prolog for the registers belonging to its DKR except for the registers that are saved already. Fig. 8 shows prolog codes generated at each node. For example, at node 1, all registers in the corresponding DKR, r27 and r28, are saved. At node 2, the DKR contains r28 which is already saved

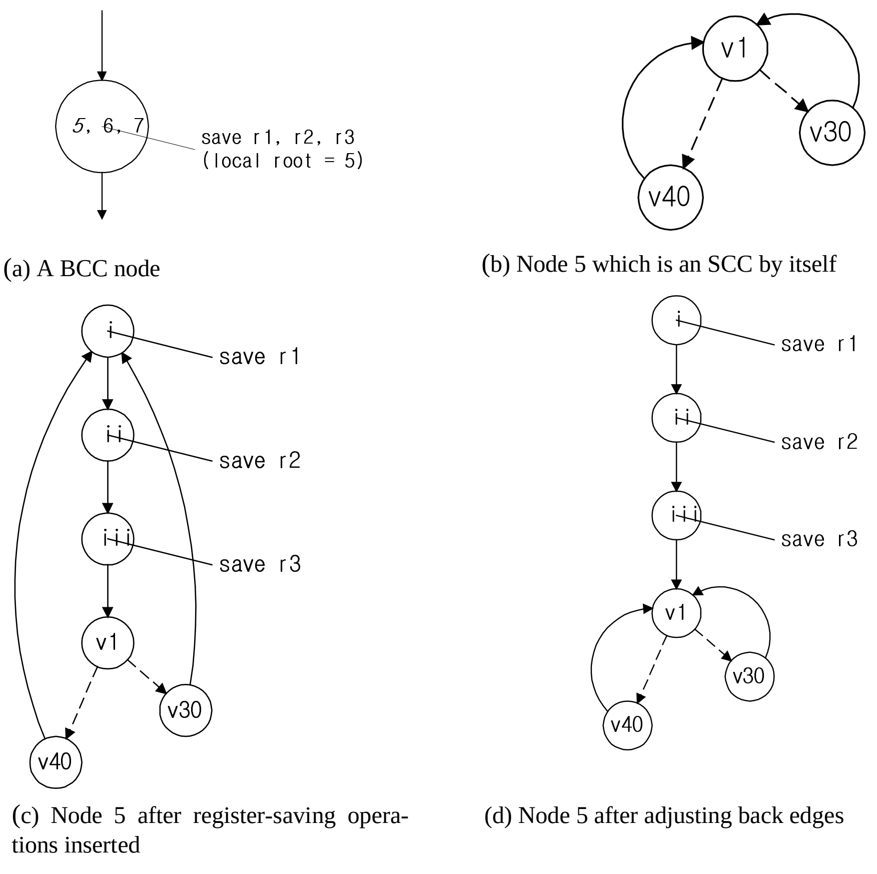
If we have to insert register-saving operations inside an SCC, we need an extra step as in Fig. 9. Fig. 9(a) shows a BCC node that includes node 5, 6, and 7. We assume that node 5 is by itself an SCC, and that it is the entry point of this BCC. Fig. 9(b) shows how node 5 looks like. When the algorithm decides that a prolog has to be generated at this BCC, the actual register-saving operations are generated on the entry node of it, which is node 5. Since node 5 is an SCC, the operations are generated on the starting basic block of this SCC, which currently includes only v1 as shown in Fig. 9(b). After inserting the register-saving operations, the flow graph becomes Fig. 9(c). Now the problem is clear: the register-saving operations are inside a loop.

We need to adjust the targets of node v30 and v40, the children of v1, so that the prolog is hoisted out of the loop. The overall prolog generation algorithm is in Fig. 10.

5. Experiments

To measure the performance of our prolog tailoring algorithm, we took 8 procedures from xlisp 2.1, performed prolog tailoring on them, counted how many register-saving operations are generated, and finally computed the reduction rates compared to the numbers without prolog tailoring. Assuming all paths are selected equally, the average number of register-saving operations per path can be computed as follows.

$$AS = \sum_{i=1}^{NE} \frac{PE_i \cdot NS_i}{PT}$$



(a) A BCC node

(b) Node 5 which is an SCC by itself

(c) Node 5 after register-saving operations inserted

(d) Node 5 after adjusting back edges

Fig. 9. Generating register-saving operations inside an SCC node

```
insert_regsave_code(node_t *n, regset_t tosave)
{
    generate register-saving operations for registers in "tosave" in the beginning of "n";
    if( "n" is an SCC byitself ) {
        old_start ← the location after the generated operations;
        for( all branching operations in "n" )
            if( branching to "n" )
                adjust to branch to old_start;
    }
}
```

```
insert_prolog(bfgnode_t *n)
{
    if(there are registers in DKR(n) that are not saved yet)
    {
        v ← the registers in DKR(n) that are not saved yet;
        for( all local root nodes of "n", k )
            insert_regsave_code(k, v) ;
    }
    for( all children of "n", j)
        insert_prolog(j);
}
```

Fig. 10. Algorithm for generating register-saving instructions

In above, PT is the number of executable paths; NE, the number of exit points; NS_i, the number of register-saving operations on a path ending with exit point *i*; PE_i, the number of possible paths reaching to exit point *I*; and finally, AS, the average number of register-saving operations.

The result in Table 2 shows that the average reduction rate is 17.4%. Excluding the most and the least reduction rates, it is 12.82%.

Table 2. The decreased number of register save instructions by prolog tailoring

procedure	Before tailoring	After tailoring	difference	Reduction rate(%)
placeform	7	4.51	2.49	35.57
mark	8	5.50	2.50	31.25
sweep	9	6.50	2.50	27.78
xlpatprop	5	4.00	1.00	20.00
evlist	9	7.50	1.50	16.67
xlenter	6	5.79	0.21	3.50
evalh	9	8.70	0.30	3.33
cons	9	8.85	0.15	1.67
average				17.47
Normalized average				12.82

6. Conclusion

In this paper, we have proposed a prolog tailoring technique to reduce the overhead of prolog code in a procedure. Our algorithm generates register-saving operations as close as possible to the actual destruction points of the corresponding registers, but without unnecessary duplication and without incorrect modification of the original program's semantics. To achieve this, the proposed algorithm transforms the given control flow graph of a procedure into a BCC flow graph, compute DKRs on it, and generates prolog code. Through experimentations, we have observed that our method reduces the number of register-saving operations by 12.82% in average.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers*, Addison Wesley, 1988

[3] F. E. Allen, and J. Cocke, "A Catalogue of Optimizing Transformations," In *Design and Optimization of Compilers*, Prentice-Hall, 1972

[4] K. Ebcioglu, R. D. Groves, K. C. Kim, G. M. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," *ACM SIGPLAN Not.*, Vol.29, No.6, pp.36-48, 1994

[5] J. Huang and D.J.Lilja, *Extending Value Reuse to Basic Blocks with Compiler Support,* IEEE Transactions on Computers, Vol. 49, No. 4, April 2000.

[6] M. Lipasti, C. Wilkerson, and J. Shen, Value Locality and Load Value Prediction, Proc. Eighth Int'l Conf. Architecture Support for Programming Languages and Operating Systems, Oct. 1996.

[7] M. Lipasti and J. Shen, Exceeding the Dataflow Limit via Value Prediction, Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture, Dec. 1996.

[8] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced Compiler Technology for the RISC System/6000 Architecture," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990

[9] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Computing*, Vol.1, No.2, pp.146-160, 1972

[10] G. Tyson and T. Austin, Improving the Accuracy and Performance of Memory Communication through Renaming, Proc. 30th Ann. Int'l Symp. Microarchitecture, Dec. 1997.

Appendix A. An Example

To show how the prolog tailoring algorithm is working, we took an example procedure from the source code of xlisp 2.1: placeform() in xlcont.c. Fig. A(a) shows the basic block control flow graph of this procedure. The graph is obtained from the assembly code produced by the IBM RS/6000 xlc compiler; therefore it is already epilog-tailored. Node 1 is the entrance point, and node 20, 29, 34, 42, 48, and 50 are the split exit points.

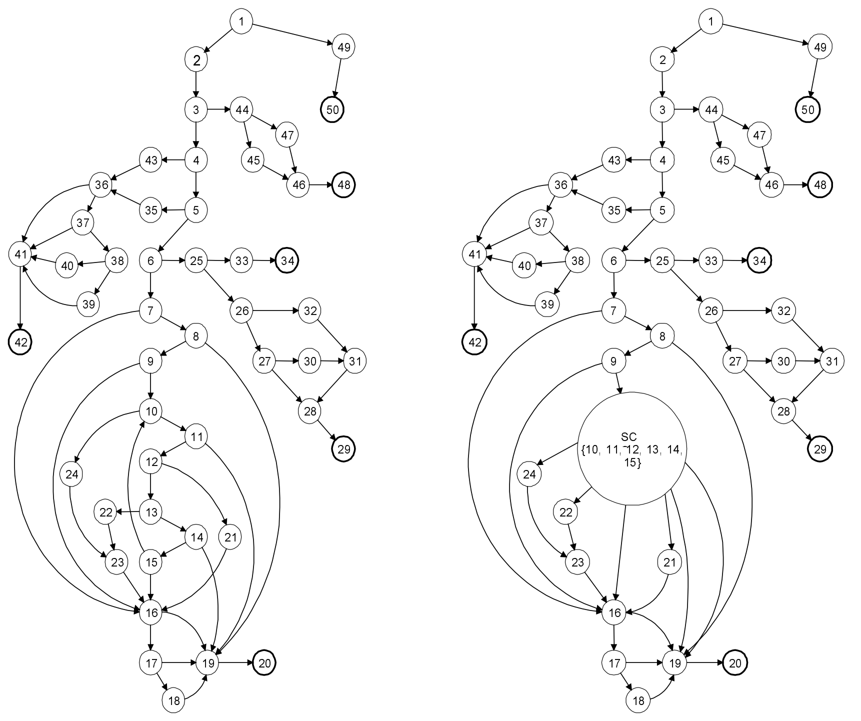


Fig. A. Control flow graph of *placeform*

A.1. Removing loops to generate an SCC flow graph

Tarjan's algorithm is applied to detect and remove SCCs. Node 10, 11, 12, 13, 14, and 15 form an SCC: they are replaced with a single node as in Fig. A(b). The incoming to and outgoing edges from these nodes are adjusted accordingly.

A.2. Removing diamond structures to generate a BCC flow graph

The next step is to remove diamond structures. Tarjan's BCC detection algorithm is used to find out all BCCs. Those nodes shared by more than one BCC are removed by the algorithm in Fig. 6. Table A1 shows the process of removing shared nodes. The second column in the table shows the BCCs detected by Tarjan's algorithm. The boldfaced nodes are the entry nodes (or local root nodes) in each BCC.

To remove shared nodes, we apply the algorithm in Fig. 6. First step is to remove all local root nodes that have outgoing edges from the current BCC. The local root nodes of BCC b2, b3, b6, b7, b13, b14, b15, b16, and b18 have outgoing edges; thus removed. After this removal, we search for shared nodes. The one in the preceding BCC node will be removed. Node 2 in b2, node 49 in b3, node 4 in b6, node 44 in b7, node 46 in b8, node 36 in b10, node 41 in b11, node 33 in b16, node 26 in 18, node 28 in b19, and finally node 19 in b21 are thus all removed. Lastly, node 1, the root of the SCC flow graph, does not exist in the final BCC set; therefore, we generate a BCC with node 1 as the only member and make it the root of the BCC flow graph, too. The third column in Table A1 shows the final BCC set. After adding proper edges, the final BCC flow graph is built as in Fig. B.

Table A1. Differences between before and after removing shared nodes

BCC #	BCCs	BCCs after removing shared nodes
b1	-	**1**
b2	**1**, 2	-
b3	**1**, 49	-
b4	**49**, 50	**49**, 50
b5	**2**, 3	**2**, 3
b6	**3**, 4	-
b7	**3**, 44	-
b8	**44**, 45, 46, 47	**44**, 45, 47
b9	**46**, 48	**46**, 48
b10	**4**, 5, 43, 35, 36	**4**, 5, 35, 43
b11	**36**, 37, 38, 39, 40, 41	**36**, 37, 38, 39, 40
b12	**41**, 42	**41**, 42
b13	**5**, 6	**6**
b14	**6**, 7	-
b15	**6**, 25	**25**
b16	**25**, 33	-
b17	**33**, 34	**33**, 34
b18	**25**, 26	-
b19	**26**, 27, 28, 30, 31, 32	**26**, 27, 30, 31, 32
b20	**28**, 29	**28**, 29
b21	**7** … 19, 21 … 24	**7** … 18, 21 … 24
b22	**19**, 20	**19**, 20

A.3 Computing DKRs and generating register-saving operations

Killed-registers are computed for the BCC flow graph as in Table A2. They are shown in the second column in Table A2. Based on the formula in Section 4.3, DKRs are computed as shown in the third column of Table A2. With these DKRs, we can determine where to save which registers as in the fourth column of the table.

Table A2. Killed registers and DKR

BCC #	Killed registers	DKR	Registers to be saved
b1	r28, r29, r31, lr	r28, r29, r31, lr	r28, r29, r31, lr
b4	r30, r31, lr	r30, r31, lr	r30
b5	r31	r31, lr	-
b8	r30, r31, lr	r30, r31, lr	r30
b9	-	-	-
b10	-	lr	-
b11	r30, lr	r30, lr	R30
b12	-	-	-
b13	-	lr	-
b15	-	lr	-
b17	lr	lr	-
b19	r27, r28, r29, r30, r31, cr4, lr	r27, r28, r29, r30, r31, cr4, lr	r27, r30, cr4
b20	-	-	-
b21	lr	lr	-
b22	-	-	-

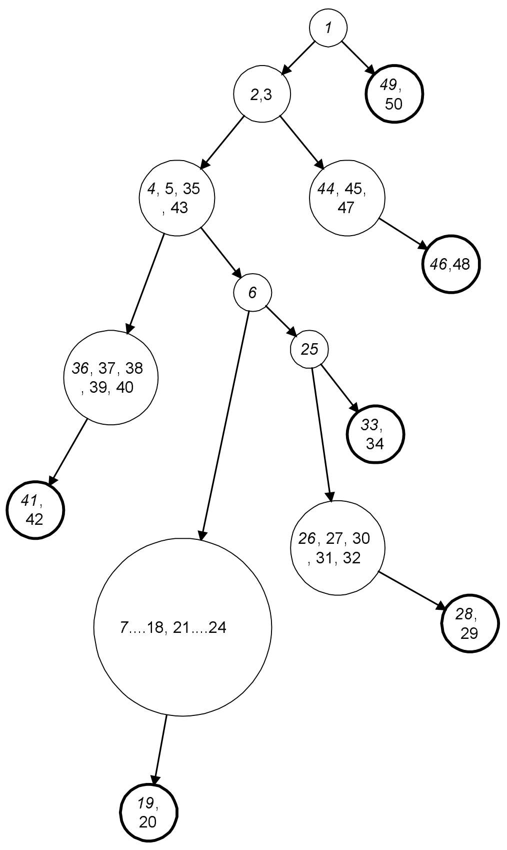


Fig. B. A BCC flow graph for *placeform()*

Hierarchical Constraint Satisfaction Based on Subdefinite Models

Dmitry Ushakov

Ledas Ltd., Acad. Lavrentjev ave., 6, Novosibirsk, 630090, Russia,
ushakov@sib3.ru

Abstract. Hierarchical Constraint Satisfaction Problems (HCSPs) are at the centre of attention in the fields of computer aided design and user interfaces. Till recently, the algorithms proposed to solve the problems of this class focused only on its small subclasses (like problems with acyclic constraint graph or linear systems). Here we present a new family of hierarchical constraint satisfaction algorithms based on the mechanism of subdefinite models. The main advantage of the proposed algorithms is their applicability to a broad class of problems, including cyclic and non-linear ones.

1 Preliminaries

The mechanism of *subdefinite models* was proposed by the Russian scientist Alexander Narin'yani at the beginning of 1980s [1] independently on the western works in the field of constraint satisfaction problems (CSPs) [2]. Today we can say that this mechanism is a general-purpose apparatus to deal with CSPs. Using it, we have no restrictions on the domain of variables (taking into account both finite and continuous ones), the nature of constraints (dealing with both binary and n-ary constraints, implicit and explicit ones). The modern description of the mechanism of subdefinite models as well as the proof of its correctness can be found in [3]. We use the well-known notion of many-sorted algebraic models to feel ourselves freely in discussing general properties of the algorithms under consideration (and thus to apply our results to a broad class of problems, including finite, continuous and mixed ones).

With this chapter, we redefine the notion of HCSP (that was firstly proposed by Borning et al. [4]) in many-sorted terms.

1.1 Hierarchical Constraint Satisfaction Problem

A *many-sorted signature* is a triple $\Sigma = (S, F, P)$ where

- S is a set of *sorts* (elements of S are names of different domains); we denote the set of all chains of elements of S (including an empty chain, λ) by S^*, i. e. $S^* = \{\lambda\} \cup S \cup S^2 \cup \ldots$, we also define $S^+ = S^* \setminus \{\lambda\}$;
- F is an $(S^* \times S)$-indexed family of sets of *operators (function symbols)* (i. e. $F = \{F_{w,s} \mid w \in S^*, s \in S\}$); $F_{\lambda,s}$ is called the set of *constants* of sort s;

– $P = \{P_w \mid w \in S^+\}$ is a family of *predicate symbols* containing the predicate symbol of equality, $= \in P_{ss}$, for each sort $s \in S$.

Let $\Sigma = (S, F, P)$ be a many-sorted signature and X be an S-sorted set (of *variables*) such that $X_{s'} \cap X_{s''} = \emptyset$ for $s' \neq s''$, and $X_s \cap F_{\lambda,s} = \emptyset$ for any $s \in S$. We define $\Sigma(X)$-*terms* as the smallest S-indexed set $T_\Sigma(X)$, such that

– $X_s \subseteq T_\Sigma(X)_s$ and $F_{\lambda,s} \subseteq T_\Sigma(X)_s$ for all $s \in S$;
– if $f \in F_{w,s}$ and $t_i \in T_\Sigma(X)_{s_i}$ for $w = s_1 \ldots s_n \in S^+$, then the string $f(t_1, \ldots, t_n)$ belongs to $T_\Sigma(X)_s$.

We define a $\Sigma(X)$-*constraint* c as an atom $p(t_1, \ldots, p_n)$, where $p \in P_{s_1 \ldots s_n}$, $t_i \in T_\Sigma(X)_{s_i}$ $(i = \overline{1,n})$. We denote the set of all variables occurring in constraint c by $\mathrm{var}(c)$.

A Hierarchical Constraint Satisfaction Problem (*HCSP*) over signature Σ is a pair (X, C), where X is a set of variables, and $C = \{C_0, C_1, \ldots, C_m\}$ is a family of finite sets of $\Sigma(X)$-constraints. A constraint $c \in C_0$ is called *a required constraint*; constraints from $C_1 \cup \ldots \cup C_m$ are called *non-required* ones.

1.2 Semantics

For a many-sorted signature $\Sigma = (S, F, P)$, a *many-sorted* Σ-*model* M is a first-order structure consisting of

– a family of *carriers* s^M for all $s \in S$ (for $w = s_1 \ldots s_n$ we denote the Cartesian product $s_1^M \times \ldots \times s_n^M$ by w^M),
– a family of *functions* $f^M : w^M \to s^M$ for all $f \in F_{w,s}$, if $w = \lambda$, then $f^M \in s^M$,
– a family of *predicates* $p^M \subseteq w^M$ for all $p \in P_w$, the predicate of equality $=^M \subseteq (ss)^M$ is $\{(a, a) \mid a \in s^M\}$ for all $s \in S$.

Let $\Sigma = (S, F, P)$ be a many-sorted signature, X be an S-sorted set of variables, M be a Σ-model. We call any S-sorted function[1] $\theta : X \to M$ an *estimate* of variables X in Σ-model M. The *extension* of the estimate θ to the set $T_\Sigma(X)$ is a function $\theta^* : T_\Sigma(X) \to M$ defined as follows. For $t \in T_\Sigma(X)_s$, $s \in S$:

$$\theta_s^*(t) = \begin{cases} \theta_s(x), & \text{if } t \equiv x \text{ for any } x \in X_s, \\ c^M, & \text{if } t \equiv c \text{ for any } c \in F_{\lambda,s}, \\ f^M(\theta_{s_1}^*(t_1), \ldots, \theta_{s_n}^*(t_n)), & \text{if } t \equiv f(t_1, \ldots, t_n) \text{ for } f \in F_{w,s}, \\ & w = s_1 \ldots s_n \in S^+, t_i \in T_\Sigma(X)_{s_i}. \end{cases}$$

Let M be a Σ-model. We will say that a constraint $c \equiv p(t_1, \ldots, t_n)$ *holds* in the model M iff there exists an estimate $\theta : X \to M$ of the variables X in the model

[1] An S-sorted function $\theta : X \to M$ is actually a family of mappings

$$\theta = \{\theta_s : X_s \to s^M \mid s \in S\}.$$

M, s. t. $(\theta^*_{s_1}(t_1), \ldots, \theta^*_{s_n}(t_n)) \in p^M$, where $\theta^* : T_\Sigma(X) \to M$ is the extension of the estimate θ to the set of $\Sigma(X)$-terms. In this case we will also say that c *holds on* θ, or θ *satisfies* c.

Let M be a Σ-model. For an HCSP $P = (X, C)$ we define the set $S^M_{P,0}$ of its *basic solutions* in the model M as follows:

$$S^M_{P,0} = \{\theta : X \to M \mid (\forall c \in C_0)\, c \text{ holds on } \theta\}.$$

The set of basic solutions thus is the set of all estimates, which satisfy the required constraints. For non-required constraints of level $i > 0$ we define the set $S^M_{P,i}$ as follows:

$$S^M_{P,i} = \{\theta : X \to M \mid (\forall j = \overline{0,i})(\forall c \in C_j)\, c \text{ holds on } \theta\}.$$

(Therefore, $S^M_{P,0} \supseteq S^M_{P,1} \supseteq \ldots \supseteq S^M_{P,m}$.) Real-world HCSPs are often over-constrained. It means that often not only $S^M_{P,m} = \emptyset$, but $S^M_{P,1}$ can be empty too. Therefore, we need a theory to choose what basic solution does better satisfy non-required constraints. In other words, we need to compare basic solutions with respect to non-required constraints. A predicate $\text{better}^M_P \subseteq S^M_{P,0} \times S^M_{P,0}$ is called *a comparator* if it has the following properties for any $\theta, \phi, \psi \in S^M_{P,0}$:

1. *Irreflexivity*: $\neg\text{better}^M_P(\theta, \theta)$
2. *Transitivity*: $\text{better}^M_P(\theta, \phi) \wedge \text{better}^M_P(\phi, \psi) \Rightarrow \text{better}^M_P(\theta, \psi)$
3. *Correctness*: $(\forall i > 0)\ \theta \in S^M_{P,i} \wedge \phi \notin S^M_{P,i} \Rightarrow \text{better}^M_P(\theta, \phi)$

The set of solutions of an HCSP P in a model M w. r. t. to a comparator better^M_P is defined as follows:

$$S^M_{P,better^M_P} = \{\theta \in S^M_{P,0} \mid (\forall \phi \in S^M_{P,0})\, \neg\text{better}(\phi, \theta)\}.$$

1.3 Types of Comparators

We can classify some kinds of comparators. The simplest ones are so-called *predicate comparators*. Given a signature Σ, a Σ-HCSP $P = (X, C)$, a Σ-model M, and an estimate $\theta : X \to M$, define $H^M_{P,i}(\theta) \subseteq C_i$ as a set of constraints from C_i, which hold on θ. Using these sets we can build two comparators: lpb^M_P and gpb^M_P (these names are acronyms for *locally-predicate-better* and *globally-predicate-better*):

$$\text{lpb}^M_P(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k})\ H^M_{P,i}(\theta) \supseteq H^M_{P,i}(\phi) \\ \qquad\qquad \wedge\ H^M_{P,k}(\theta) \supset H^M_{P,k}(\phi), \end{cases}$$

$$\text{gpb}^M_P(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k})\ |H^M_{P,i}(\theta)| \geq |H^M_{P,i}(\phi)| \\ \qquad\qquad \wedge\ |H^M_{P,k}(\theta)| > |H^M_{P,k}(\phi)|. \end{cases}$$

The second group of comparators is metric ones. They are based on *an error function*. For an estimate θ and a $\Sigma(X)$-constraint c we define a non-negative real value $e^M(c,\theta)$, which is called the error of satisfaction of the constraint c on θ, and has the following property: $e^M(c,\theta) = 0 \Leftrightarrow c$ holds on θ. Given e^M, we define a comparator leb^M_{P,e^M} (an acronym for *locally-error-better*) as follows:

$$\mathrm{leb}^M_{P,e^M}(\theta,\phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall c \in C_1 \cup \ldots \cup C_k)\ e^M(c,\theta) \le e^M(c,\phi) \\ \qquad\qquad \wedge\ (\exists c \in C_k)\ e^M(c,\theta) < e^M(c,\phi). \end{cases}$$

Comparators from geb (*globally-error-better*) family take into account global information about errors on each level. They can be expressed using a global error $g_{e^M}(C_i,\theta) = g(e^M, C_i, \theta)$, which summarizes all the errors of constraints from C_i on the estimate θ:

$$\mathrm{geb}^M_{P,e^M,g}(\theta,\phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k})\ g_{e^M}(C_i,\theta) \le g_{e^M}(C_i,\phi) \\ \qquad\quad \wedge\ g_{e^M}(C_k,\theta) < g_{e^M}(C_k,\phi). \end{cases}$$

We will use two global error functions: wc (an acronym for *worst-case*), and ls (*least-squares*):

$$\mathrm{wc}(e^M, C_i, \theta) = \max_{c \in C_i} e^M(c,\theta),$$
$$\mathrm{ls}(e^M, C_i, \theta) = \sum_{c \in C_i} (e^M(c,\theta))^2.$$

We will use notations wcb^M_{P,e^M} and lsb^M_{P,e^M} for geb-comparators based on wc and ls functions respectively.

Note, that predicate comparators can be unified with error ones by introducing a special error function pe, called *predicate error*:

$$\mathrm{pe}^M = \begin{cases} 0, & \text{if } c \text{ holds on } \theta, \\ 1, & \text{otherwise.} \end{cases}$$

Then lpb and gpb comparators can be expressed via leb and lsb ones respectively. Therefore, it is sufficient to consider only three comparators: leb, wcb, and lsb. However, one can propose simpler algorithms for hierarchical constraint satisfaction based on predicate comparators rather than on error ones.

1.4 A Simple Example

Consider the following simple example of hierachical constraint satisfaction problem. Imagine an end-user who draws and moves some figures on a screen, put positioning constraints on them, and a system which automatically redraws the figures on the screen according to user constraints. The user drew a point at position $(0,0)$ and put a required constraint that this point cannot be moved. Then he drew another point at position $(100,100)$ and put a strong constraint of distance between two these points: "the distance between them is not less than 10". Then he moves the second point to some position (x,y) on the screen. What is a possible reaction of the system? We can easily specify this problem as an

```
Fix, Var: Point2D;
required: Fix.x = 0; Fix.y = 0;
strong:   Distance( Fix, Var ) >= 10;
medium:   Var.x = x; Var.y = y;
weak:     Var.x = 100;   Var.y = 100;
```

Fig. 1. Moving a Point HCSP

HCSP. Here we have required constraints (the first point cannot be moved), and strong ones (distance constraint). We can model a point movement by medium constraints, and conservation of figures positions by weak constraints. Therefore, we deal with the HCSP presented in fig. 1.

Obviously, if the new coordinates of the point are outside the circle with radius 10 and center in (0,0), then the point will be moved there. Otherwise the system should firstly satisfy strong distance constraint, and then medium one. The possible positions of the point after movement and hierarchical constraint satisfaction are presented in fig. 2.

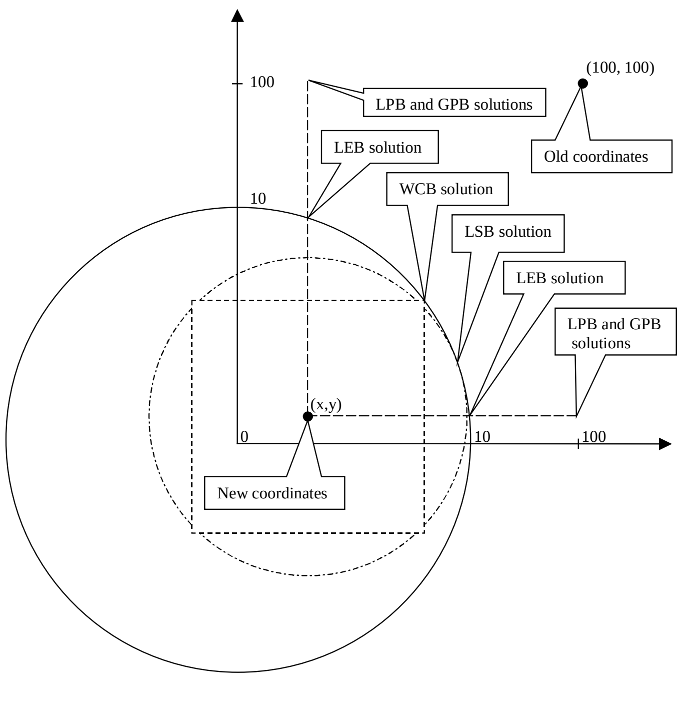


Fig. 2. Possible solutions of Moving a Point HCSP

Locally-error-better solution is produced by satisfying constraints in turn (firstly positioning x coordinate, then y, or vice versa). Since after satisfying one of these constraints we cannot satisfy another one, the weak constraints (old coordinates) is used for positioning the point on the second coordinate.

Globally-error-better solution is obtained in other way (we try to satisfy as many medium constraints as possible), but it is the same as locally-error-better one, since we cannot satisfy both coordinate constraints simultaneously.

Searching locally-error-better solution, the system tries to minimize an error of a constraint satisfaction. In our case, this error equals the difference between real and desired coordinate. A solution to be produced depends on the order of considering constraints (firstly positioning x coordinate, then y, or vice versa).

Worst-case-better solution is a positioning which minimizes the maximal deviation of one of coordinates from desired value. Semantics of the solution can be expressed by the square of minimal size with center in (x, y), which contacts with the big circle. The point of contact is a solution.

Similarly, least-squares-better solution can be expressed by the circle of minimal size with center in (x, y), which contacts with the original circle.

2 Subdefinite Models

The algorithms of hierarchical constraint satisfaction, discussed below, are implemented in *subdefinite models* framework. Before considering the algorithms, we briefly remind the basic concepts of subdefinite models.

2.1 Subdefinite Extensions

In [3] we have shown that subdefinite model approach allows one to find an approximation of the set of all solutions of a CSP (in terminology of this paper, to find an approximation of the set of all basic solutions of an HCSP). This approximation is done by the means of achieving local subdefinite consistency. First, we build *subdefinite extensions* (*SD-extensions*) of universes (carriers) of given Σ-model. If U is a universe, then its subdefinite extension, ${}^*U$ is a set of subsets of U, satisfying the following properties:

1. $\{\emptyset, U\} \subseteq {}^*U$.
2. $(\forall\, V, W \in {}^*U)\; V \cap W \in {}^*U$.
3. There are no infinite decreasing chains $(V \supset W \supset \ldots)$ in ${}^*U$.

Any subset V of U can be approximated in SD-extension ${}^*U$ as follows:

$$\mathrm{app}_{{}^*U}(V) = \bigcap_{V \subseteq W \in {}^*U} W. \tag{1}$$

Let X be an S-sorted set of variables, M be a Σ-model, and ${}^*M$ be its SD-extension[2]. *A subdefinite estimate* (*SD-estimate*) is an S-sorted mapping

$$\Theta = \{\Theta_s : X_s \to {}^*s^M \mid s \in S\},$$

which maps each $x \in X_s$ $(s \in S)$ into *a subdefinite value* $\Theta(x) \in {}^*s^M$. An SD-estimate Θ is *narrower* than another SD-estimate Φ iff $\Theta(x) \subseteq \Phi(x)$ for all $x \in X_s$, $s \in S$. An estimate θ is contained in an SD-estimate Θ (writing $\theta \in \Theta$) iff $\theta(x) \in \Theta(x)$ for all $x \in X_s$, $s \in S$. An SD-estimate, which does not contain any estimate, is called an empty SD-estimate (writing $\Theta = \emptyset$).

Given a signature Σ, and an S-sorted set of variables X, let c be a Σ-constraint, M be a Σ-model, and ${}^*M$ be its SD-extension. *A filtering* $\mathcal{F}_c$ of the constraint c is a mapping on the set of SD-estimates, satisfying the following conditions for any SD-estimates Θ, Φ:

1. *Contractness*: $\mathcal{F}_c(\Theta) \subseteq \Theta$.
2. *Monotonicity*: $\Theta \subseteq \Phi \Rightarrow \mathcal{F}_c(\Theta) \subseteq \mathcal{F}_c(\Phi)$.
3. *Correctness*: if c holds on θ, then $\theta \in \Theta \Rightarrow \theta \in \mathcal{F}_c(\Theta)$.
4. *Idempotency*: $\mathcal{F}_c(\mathcal{F}_c(\Theta)) = \mathcal{F}_c(\Theta)$.

An SD-estimate Θ is *consistent* w. r. t. a constraint c iff $\mathcal{F}_c(\Theta) = \Theta$. It is easy to see, that for any set C of constraints there exists unique SD-estimate Θ^*_C s. t.:

1. Θ^*_C is consistent w. r. t. each $c \in C$.
2. If an SD-estimate Φ is consistent w. r. t. each $c \in C$, then $\Phi \subseteq \Theta^*_C$.

Moreover, if there exists an estimate θ s. t. each $c \in C$ holds on θ, then $\theta \in \Theta^*_C$. Therefore, Θ^*_C is an approximation of the set of all the estimates which satify any $c \in C$. For an HCSP $(X, \{C_1, \dots, C_m\})$, $\Theta^*_{C_0}$ is an approximation of the set of its basic solutions.

Fig. 3 shows the algorithm of finding the maximal consistent SD-estimate for a given set of constraints C. It uses a global structure `Ass`, where `Ass`(x) is a set of constraints from C where the variable x occurs. The function returns `False` if the inconsistency is detected during filtering (it means $\Theta^*_C = \emptyset$). Choosing c in Q (the fourth line) can be arbitrary, but we use the principle "first in — first out", i. .e. regard Q as a queue. In [3] we have proved that the call `Revise`(Θ^0, C), where $\Theta^0(x) = s^M$ for any $x \in X_s$, $s \in S$ always produces Θ^*_C.

2.2 Solving an HCSP

We deal with a signature $\Sigma = (S, F, P)$, where there is a sort real $\in S$, and all constant, functional, and predicate symbols on it. We also deal with a Σ-model M, where real^M is the set of all real numbers, and all functional and predicate symbols have traditional interpretation ("+" as addition, "=" as equality, etc.)

[2] For the purpose of this paper we define an SD-extension of a Σ-model M as a set of SD-extensions of its carriers $\{{}^*s^M \mid s \in S\}$. Here we deal no with SD-extensions of functions and predicates.

```
function Revise( in out $\Theta$, in $Q$ ) : boolean
begin
  while $Q \neq \emptyset$ do
    choose $c \in Q$;
    $\Phi \leftarrow \mathcal{F}_c(\Theta)$;
    for $x \in \mathrm{var}(c)$ do
      if $\Theta(x) \neq \Phi(x)$ then
        if $\Theta(x) = \emptyset$ then return False end if;
        $Q \leftarrow Q \cup \mathtt{Ass}(x)$
      end if
    end for;
    $Q \leftarrow Q \setminus \{c\}$;
    $\Theta \leftarrow \Phi$
  end while;
  return True;
end.
```

Fig. 3. The algorithm for achieving the maximal consistency

Suppose that all non-required constraints look like $x = \overline{0}$, where $x \in \mathrm{real}^M$, and $\overline{0} \in F_{\lambda,\mathrm{real}}$ with standard interpretation $\overline{0}^M$ as the real zero. (Below we consider the transformation of arbitrary HCSP to this form.) The need of globally processing all the constraints of the same level suggests us to deal with one constraint per level. Such a constraint has a form $\mathrm{zero}(x_1, \ldots, x_n)$ and is the reduction of a group of constraints $\{x_1 = \overline{0}, \ldots, x_n = \overline{0}\}$. It is reasonable to use the same schema of calculations for all the types of comparators. This schema is presented in fig. 4. The call `FilterNonRequiredConstraint` stands for the one of the procedures of filtering presented in fig. 5: `FilterLPB`, `FilterGPB`, `FilterLEB`, `FilterWCB`, or `FilterLSB`. Moreover, one can use different versions of comparators on each level of constraint hierarchy.

Procedure `FilterLPB` has nothing special, but others use an internal stack to implement the depth-first search of the solution. If the inconsistency is detected in some branch, the procedure returns to the previous suspended branch.

Procedure `FilterLEB` tries to narrow an SD-value of an argument variable as closer as possible to zero. However it has the following drawback. Suppose we have a non-required constraint $x = \overline{0}$ and required one $x = y - z$. (In fact, it means we have a non-required constraint $y = z$.) Suppose there is no basic solution θ with $\theta(x) = 0$. When we try to narrow an SD-value of x to zero, we need to assign x with some value $\mathrm{app}_{{}^*\mathrm{real}^M}(\{a\})$, where $a > 0$. Roughly speaking, it means we try to filter a constraint $|y - z| = a$. This constraint has a disjunctive nature, since we do not know what constraint should be satisfied: $y = z + a$ or $z = y + a$. This lack of knowledge is often the reason of poor filtering.

```
algorithm SolveHCSP( in $P = (X, \{C_0, \{c_1\}, \ldots, \{c_m\}\})$, out $\Theta$ )
begin
  [ build structure Ass ];
  for $s \in S$, $x \in X_s$ do
    $\Theta(x) \leftarrow s^M$   % assigning the maximal undefined values
  end for;
  if not Revise($\Theta, C_0$) then
    $\Theta \leftarrow \emptyset$
  else
    for $i = 1, \ldots, m$ do
      FilterNonRequiredConstraint($\Theta, c_i$)
    end for
  end if
end.
```

Fig. 4. The general algorithm for solving an HCSP

The `FilterWCB` and `FilterLSB` procedures differ only in the function $g(\Theta)$:

$$g_{\text{wcb}}(\Theta) = \max_{i=1}^{n} \sup \Theta(x_i),$$
$$g_{\text{lsb}}(\Theta) = \sum_{i=1}^{n} (\sup \Theta(x_i))^2\}.$$

One can note that our algorithms are not complete in general sense: we cannot guarantee the existence of a solution in resulted subdefinite values. However, in real-world problems we can easily add weakest non-required constraints $x = a_x$ (with arbitrary chosen a_x) for *all* the variables of an HCSP under consideration, and thus the resulted values will be defined.

2.3 Transformation of an HCSP to a Simple Form

Remember, all the algorithms above deal with an HCSP, where all non-required constraints look like $x = \overline{0}$ for real variable x. How to transform any HCSP to this form? First, we need to extend our signature $\Sigma = (S, F, P)$ with a functional symbol diff_s for each sort $s \in S$: $\text{diff}_s \in F_{ss,\text{real}}$. This symbol should have the following interpretation in a Σ-model M: $\text{diff}_s^M(a, b) = 0 \Leftrightarrow a = b$. For example, $\text{diff}_{\text{real}}$ can be implemented as $\text{diff}_{\text{real}}^M(a, b) = |a - b|$.

Consider a constraint $p(t_1, \ldots, t_n) \in C_k$ $(k > 0)$, where p is a predicate symbol, and $t_i \in T_\Sigma(X)_{s_i}$ $(i = \overline{1, n})$ are terms. Let $u_1, \ldots, u_n$ be new variables of sorts $s_1, \ldots, s_n$ respectively, and $v_1, \ldots, v_n$ be new variables of sort real. Then we transform our constraint into a set of ones:

– required constraint $p(u_1, \ldots, u_n)$,
– required constraints $\text{diff}_{s_i}(u_i, t_i) = v_i$ for $i = \overline{1, n}$,
– non-required constraints $v_i = \overline{0}$ for $i = \overline{1, n}$.

```
procedure FilterLPB( in out $\Theta$, in $c \equiv \mathrm{zero}(x_1, \dots, x_n)$ )
begin
  for $i = \overline{1, n}$ do
    if $0 \in \Theta(x_i)$ then
      $\Phi \leftarrow \Theta$;
      $\Phi(x_i) \leftarrow \mathrm{app}_{{}^*\mathrm{real}^M}(\{0\})$;
      if Revise($\Phi$, Ass($x_i$)) then $\Theta \leftarrow \Phi$ end if
    end if
  end for
end.
```

```
procedure FilterGPB( in out $\Theta$, in $c \equiv \mathrm{zero}(x_1, \dots, x_n)$ )
begin
  $\Theta^* \leftarrow \Theta$; $k^* \leftarrow 0$;
  push($\Theta, 1, 0$);
  while non-empty-stack do
    pop($\Theta, i, k$);
    if $i > n$ then if $k > k^*$ then $\Theta^* \leftarrow \Theta$; $k^* \leftarrow k$ end if
    else if $0 \in \Theta(x_i)$ then
      push($\Theta, i + 1, k$);
      $\Theta(x_i) \leftarrow \mathrm{app}_{{}^*\mathrm{real}^M}(\{0\})$;
      if Revise($\Theta$, Ass($x_i$)) then push($\Theta, i + 1, k + 1$) end if
    end if end if
  end while;
  $\Theta \leftarrow \Theta^*$
end.
```

```
procedure FilterLEB( in out $\Theta$, in $c \equiv \mathrm{zero}(x_1, \dots, x_n)$ )
begin
  $i \leftarrow 1$;
  while $i \le n$ do
    push($\Theta, i$);
    $\Theta(x_i) \leftarrow \mathrm{app}_{{}^*\mathrm{real}^M}(\{\inf |\Theta(x_i)|\})$;
    if not Revise($\Theta$, Ass($x_i$)) then pop($\Theta, i$); end if
    $i \leftarrow i + 1$
  end while
end.
```

```
procedure FilterWCB/LSB( in out $\Theta$, in $c \equiv \mathrm{zero}(x_1, \dots, x_n)$ )
begin
  $w^* \leftarrow 0$;
  repeat
    $w \leftarrow g(\Theta)$;
    push($\Theta, \frac{w^*+w}{2}$);
    for $i = \overline{1, n}$ do $\Theta(x_i) \leftarrow \Theta(x_i) \cap [-\frac{w^*+w}{2}, \frac{w^*+w}{2}]$ end for
    if not Revise($\Theta$, $\cup_{i=\overline{1,n}}$Ass($x_i$)) then pop($\Theta, w^*$); end if
  until $w - w^* < \varepsilon$   % precision of calculation
end.
```

Fig. 5. The procedures of filtering better solution

2.4 Implementation Issues

These algorithms have been implemented in the framework of constraint programming environment *NeMo+* [6]. Table 1 presents CPU time measured on AMD Athlon 1.1 GHz Windows NT Workstation when solving two HCSPs on different input data. `Point` (x,y) denotes *Moving a Point HCSP* considered

```
A, B, C: Point2D;
required: Angle( A, C, B ) = Pi/2;
strong:   Distance( B, C ) <= Distance( A, C );
medium:   Distance( B, C ) = x;
          Distance( A, C ) = y;
          Distance( A, B ) = z;
weak:     // constraints on coordinates...
```

Fig. 6. Right Triangle HCSP

in section 1.4 (where x and y are coordinates of new position for the point), while `Triangle` (x,y,z) denotes an HCSP presented in fig. 6. Note, we have not found lsb-solution of some problems in reasonable time (the corresponding times are marked with '?' sign in the table.

Our experiments suggest the following recommendations of using the proposed algorithms for solving HCSPs. First of all, we want to emphasize the minimal complexity of searching two kinds of solutions: locally-predicate-better and locally-error-better. We cannot say this about globally-predicate-better, since it complexity must depend on the number of constraints in each level of constraint hierarchy. Our tests are very small to do such conclusion.

The quality of worst-case-better solution suggests us to prefer it to ones discussed above, but we see, that its complexity is more greater. On the other hand, we see, that its complexity does not depend drastically on input data, and it can be small enough (less than one second for our test problems). We believe, there can exist problems, where getting this type of solution is preferable.

Discussing least-square-better solution, we need to say, that this type of solution having the best quality from all the types, however, requires a number of system resources. (In our tests, we have not obtained this solution in reasonable time for some set of input data.) But for other sets of input data its complexity is small enough. In any case, this type of hierarchical constraint satisfaction should be used carefully.

3 Related Works

There are a number of algorithms for solving an HCSP. Most of them find a locally-predicate-better solution. Among others, the two most similar to our ones

Table 1. Performance results of solution search for different HCSPs

Problem	*lpb-time*	*gpb-time*	*leb-time*	*wcb-time*	*lsb-time*
`Point (-6,8)`	0	0	40	400	30
`Point (-4,-4)`	0	0	40	410	2894
`Point (-3,1)`	0	0	40	390	3204
`Point (2,0)`	0	0	40	400	390
`Point (0,0)`	0	0	40	390	?
`Triangle (3,4,5)`	10	30	200	540	110
`Triangle (4,3,6)`	10	30	200	570	4005
`Triangle (4,3,5)`	10	20	200	530	2052
`Triangle (4,3,4)`	10	20	200	550	4286
`Triangle (4,3,2)`	10	10	200	580	?

are *Indigo* [7] and *Projection* [8]. They are both intended for searching a locally-error-better solution and deal with interval values of variables. *Indigo* processes acyclic constraint graphs with numerical constraints and has the polynomial complexity. *Projection* processes systems of linear equations and inequalities but takes exponential time in the worst case. Of course, our general-purpose algorithm is not so efficient as these two, but it can be applied to non-linear systems with cyclic constraint graph: this is its main advantage.

Acknowledgements. The author is indebted to anonymous referees for their valuable comments.

References

1. Narin'yani, A.S.: Subdefiniteness and Basic Means of Knowledge Representation. Computers and Artificial Intelligence, Bratislawa **2(5)** (1983) 443–452
2. Mackworth, A.K.: Consistency in Networks of Relations. Art. Int. **8** (1977) 99–118
3. Ushakov, D.: Some Formal Aspects of Subdefinite Models. Preprint of A. P. Ershov Institute of Informatics Systems, Novosibirsk **49** (1998) (Also available via `http://www.iis.nsk.su/preprints/USHAKOV/PREPRINT/Preprint_eng.html`)
4. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint Hierarchies. Lisp and Symbolic Computation **5** (1992) 223–270
5. Goguen, J.A., Meseguer, J.: Models and Equality for Logical Programming. LNCS **250** (1987) 1–22
6. Shvetsov, I., Telerman, V., Ushakov, D.: *NeMo+*: Object-Oriented Constraint Programming Environment Based on Subdefinite Models. LNCS **1330** (1997) 534–548
7. Borning, A., Anderson, R., Freeman-Benson, B.: Indigo: A local propagation algorithm for inequality constraints. Proc. 1996 ACM Symp. on User Interface Software and Technology (1996) 129–136
8. Harvey, W., Stuckey, P.J., Borning, A.: Compiling Constraint Solving Using Projection. LNCS **1330** (1997) 491–505

Using Constraint Solvers in CAD/CAM Systems

Vitaly Telerman

A. P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Acad. Lavrentjev ave., 630090, Novosibirsk, Russia
vitali@wanadoo.fr

Abstract. We discuss different possibilities of using the Constraint Programming Solvers (CPS) in CAD/CAM systems. The *NeMo+* CPS, based on the approach of Subdefinite Models (SD-Models), and some its specializations are considered. The paper presents some components of a CAD/CAM system, where the *NeMo+* solver is used or can be used, discusses the advantages of this approach.

Introduction

Constraints are one of fundamental things that is intuitively known for the user in all areas of activity, including the CAD/CAM one [1]. Generally speaking, each interaction of two variables can be considered as a constraint. Constraints allow the user to specify the problem in the declarative manner. He doesn't need to specify "HOW to solve the problem", but only "WHAT a problem is necessary to solve".

Obviously, it is very important HOW constraints are solved. For solving the constraint satisfaction problem in the CAD/CAM system we propose to use the object-oriented solver *NeMo+* [2, 3]. It is based on the well-known subdefinite models (SD-models) approach, proposed in the early 80th by Dr. A.S. Narin'yani [4], and founded in [5, 6].

We consider that the model of the designed entity (i.e. the designed product) in a CAD/CAM system consists of the *physical* part and the *functional* one.

The physical model is a decomposition of the product in assemblies, parts and design features. An assembly is a set of parts and or assemblies, the model can contain as many assembly levels as needed, a part is a set of features and a feature is a set of parameters that determine its properties and its behavior.

The functional model is a decomposition of the product in the different functions that it has to support. The product is divided into functions, the model can contain as many function levels as needed. Each elementary function (a function that can not be further decomposed) is implemented in the solver. A function can include features coming from different parts. The functional model allows the designer to work directly with functions, not necessarily knowing which parts are involved.

In this paper we propose an approach of a Constraint-Based CAD/CAM system, which, in our view, will have the following advantages:

- the designer has the possibility to work with the approximately known (or subdefinite) values of parameters (e.g. intervals - for real numbers, enumerations for discrete values, etc.);
- the solver returns to the designer both the validated subdefinite values, which can be more definite than the initial ones, and one of the exact solutions;
- the solver is able to solve together both the geometric constraints and the engineering ones. Thus it can considerably reduce the number of backtracks in the design process;
- the subdefinite model can be used all along the development of a design application since it can support the design knowledge acquisition, the implementation and the maintenance phases.

The paper is organized as follows. Section 1 gives a brief description of the constraint programming environment *NeMo+*. In section 2 we present a *NeMo+* specialization for solving the geometric problems. The possibilities of using the constraint solver in conceptual and assembly design is discussed in section 3. The use of *NeMo+* in Knowledge component of a CAD/CAM system is presented in section 4. Section 5 gives a brief description of the use of *NeMo+* for solving time-based problems in digital manufacturing and product data management. The last section contains the conclusions and further plans.

1 NeMo+ Constraint Programming Environment

The object-oriented constraint programming environment *NeMo+* is a state-of-the-art constraint solver that, besides the traditional constraint satisfaction algorithm, incorporates a number of constraint programming techniques: root locating, symbolic transformations and differentiation, heuristics for partial satisfaction problems, specialized module for solving geometric constraints.

The standard *NeMo+* environment has the following peculiarities:

1. An extended set of predefined data types which may have finite as well as infinite domains of values. It includes an extensive library of basic (i.e. implemented in C++) constraints for such data types as set, Boolean, integer and real, strings, and any other types defined by the user. The domain of a variable can be represented by a single value, by enumeration of possible values, by interval or multi-interval values. The choosing of data type representations allows the user to the compromise between the solution quality and the calculation time. For more details on data types in SD-models see [7, 8].

2. Availability of high-level facilities for specification of problem-oriented constraints and data types. It includes a high-level language for specification of systems of constraints. This language is a purely declarative one and allows the user to describe a system of constraints as a collection of formulas. Object-oriented properties of the *NeMo+* language are used to define the structure of a model and to define problem-oriented data types and constraints from the base ones. In addition, the language includes sophisticated means for controlling the constraint propagation process.

3. Solving the direct and the inverse problems on the same specification of the problem. Taking into account the initial values or parameters, the solver defines itself what problem should be solved (direct, inverse, or both).

4. Use of the method of subdefinite models to satisfy the system of constraints. The main feature of the method of SD-models is that it uses a single algorithm of constraint propagation to process data of different types. This allows one to solve mixed of constraints including simultaneously set, Boolean, integer and real variables and constraints on them. Moreover, *NeMo+* proposes different techniques to find an exact solution, including the optimal one.

5. *NeMo+* can process constraints defined by tables, including database ones. It chooses all reliable data from the table/database and uses them in another constraints as subdefinite data. It should be mentioned that tables/databases themselves may contain the subdefinite values [9].

The algorithm of computations implemented in SD-models is a highly parallel data-driven process [5]. Modification of the values of some variables in the common memory automatically results in calling and executing those constraints for which these variables are arguments. The process halts when the execution of constraints does not change the variables values.

Consider now an example that illustrates this algorithm. Suppose that we need to solve the following system of two linear equations:

$$y = x - 1; \qquad (F1)$$
$$2 * y = 3 * (2 - x); \qquad (F2)$$

Each equation may be regarded as an implicit function ($F1$ and $F2$) of two variables x and y. The plots of these functions are shown in Fig. 1 a).

Suppose that an initial approximation to the variable x is known: between -1 and 4 (this estimate can be represented by the interval $[-1, 4]$). The idea of subdefinite computations is to use the current estimate to compute in turn the projections of the functions $F1$ and $F2$ onto x and y. For instance, the projection of $F1$ onto y for x equal to $[-1, 4]$ is the interval $[-2, 3]$. The result of the projection is shown in Fig. 1 b).

If we now use y to compute the projection of $F2$ onto x, then we will obtain a new value of x equal to the interval $[0, 10/3]$ (see Fig. 1 c). Continuing the process, we will gradually approach the desired solution. Fig. 1 d) shows two spirals demonstrating how we approach the solution from below and from above, respectively.

It is worth noting that the parameters of real problems always have initial approximations of their values. Even if the person solving a problem cannot determine the initial constraints on the domain of some numerical parameter its subdefinite value will be estimated by the interval from minus infinity to plus infinity.

During the computation of one model, the solver starts twice. At first, it checks the consistency of the model and improves the initial subdefinite values. Then, it starts for finding one of the exact solutions. Both results, the subdefinite (consistent) values and the exact solution are persistent in the CAD/CAM system.

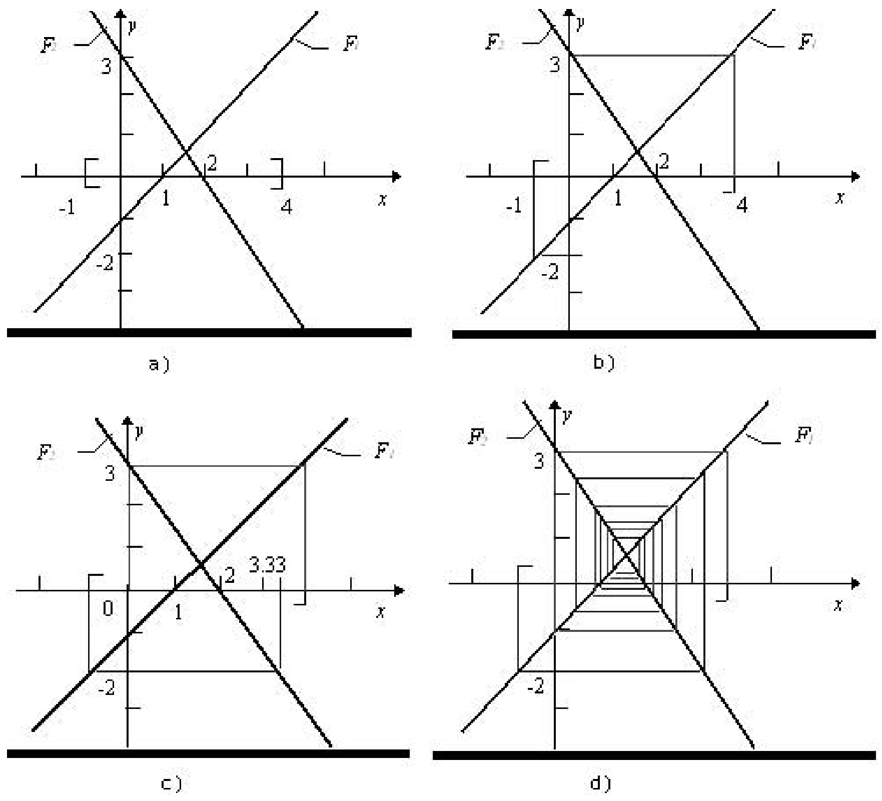


Fig. 1. Illustration of the algorithm of subdefinite computations

The *NeMo+* solver has been implemented jointly by Russian Research Institute of Artificial Intelligence (Moscow-Novosibirsk) and by Institute of Informatics Systems (Novosibirsk).

Summarizing all these properties we can say that *NeMo+* can be used in different parts of CAD/CAM systems such as:

– Sketcher (geometric solver);
– Conceptual and assembly design;
– Knowledge based engineering;
– Digital manufacturing and product data management.

Moreover, *NeMo+* can be placed in the kernel of a CAD/CAM system (so-called feature platform) to provide a general-purpose solution for both update engine and user interaction.

2 Constraint Solver for Geometric Modeler

Geometric applications in CAD/CAM area all have a fundamental requirement to maximize the productivity of the designer by enabling the efficient construction and modification of geometric models.

In our view, each geometric problem belongs to the class of constraint satisfaction problems, i.e. its specification is a declarative one, which contains a set of objects connected by a set of geometric constraints.

Present CAD systems are merely based on parametric or variational design. The best known of geometric solvers is DCM (Dimensional Constraint Manager by D-Cubed Ltd.) [10]. DCM is based on an algorithm for computing the solution for a subset of all possible equations of geometry and dimensions, using purely algebraic methods. The usual arithmetic operations are used including square roots. DCM considers in detail the equations that are obtained when points, lines and circles on a plane are defined by means of relative distance and angle constraints. The main result is that these equations can be solved algebraically for a significant class of configurations.

In [11] the DCM method is seen as a propagational solver; solving constraints that can sequentially be constructed on a drawing board, using ruler and compass. According to [11] propagational solvers offer robustness, accuracy and speed. However, they are restricted to relatively simple problems. The main problem is that mathematical constraints which determine other product characteristics than those related to geometry alone also have to be taken into account [12]. These constraints cannot easily be solved in existing CAD systems as they are highly coupled and non-linear. The second problem is the problem of measurement accuracy, and tolerancing. It is almost evident that, due to the measuring instrument's accuracy, some geometric values like lengths, distances and angles are approximately known in real-life problems. Different techniques can be used to solve such kind of problems. DCM is able to take into account the approximately known values of dimensions via inequalities, but it always returns only one exact solution.

The main advantage of *NeMo+* is that it is able to solve geometric problems with exact and/or interval values of parameters jointly with non-geometric (so-called, engineering) constraints, and it returns two kind of results: the exact solution, and the subdefinite values.

For solving the geometric problems in the standard *NeMo+* it is necessary either to specify all significant constraints (the theorem of cosine, the formulae of Heron, the sum of angles in the triangle, etc.) or to specify the problem in terms of high-level objects like triangles, rectangles, trapeze, etc, in which the coherence relationships are included yet. Obviously, both solutions are not acceptable, when *NeMo+* is used as a solver in Sketcher programs. It is more preferable that the solver make itself the decision what relationships are necessary to be taken into consideration for solving the given problem. In order to do that, a specialized library was implemented in the *NeMo+* environment, which can be considered as a *NeMo+* geometric modeler. In our view, the *NeMo+* geometric modeler is an "intelligent" instance solver, which is able to solve well-constrained, under-constrained, and over-constrained (but consistent) problems. Using the partial constraint satisfaction tecniques it also can solve the over-constrained (and inconsistent) geometric problems. The *NeMo+* geometric modeler was implemented by E.V. Roukoleev. The partial constraint satisfaction algorithms were implemented by D.M. Ushakov.

The geometric modeler allows *NeMo+* to compute the model, which contains only the elementary geometric objects (points, lines, angles, ...) and constraints (perpendicularity, parallelism, distance, ...). Using this information and the intermediary results, obtained during the constraint propagation, the modeler generates new constraints on parameters of the model. The modeler uses three methods for changing the model: unification, decomposition, and synthesis.

Unification. The basic geometric objects like points, lines and planes are considered for this method, and for each of them the concept of "index" with the following properties is determined (for objects of the same type):

1. Index(a)=Index(b)<=>a=b<=>Distance(a,b)=0; where Index: $G \to Z$.
2. If $x = F(a_1, a_2, \ldots, a_N)$, $y = F(b_1, b_2, \ldots, b_N)$, and

$$Index_F(Index(a_1), \ldots, Index(a_N)) = Index_F(Index(b_1), \ldots, Index(b_N)),$$

then $x == y$, where $Index_F : Z \times Z \times \ldots \times Z \to Z$. Thus, if in the model there are constraints of equivalence or of equality to zero of distances, the unification of the appropriate objects is done. And if the arguments of functions are unified, their results are unified too.

Decomposition. Usually the complex relations are expressed through the more simple ones. During the interpretation of the complex relation the modeler try to determine if some of its more simple components exist in the model or not. If a more simple relation exists, then the modeler doesn't create a new component and uses the existing one.

Synthesis. When we have a lot of constraints linked to the same object, sometimes it is possible to create for this object a stronger relation. For example,

```
On(PointA,LineAB)&On(PointB,LineAB)->LineAB=On(PointA,PointB);
```

In the case of three distances AB, BC and AC, this technique allows the modeler to find out the contradictions before setting up the coordinate values:

$$AB + BC >= AC;$$
$$AB + AC >= BC;$$
$$AC + BC >= AB;$$

Using this technique it is possible to solve not only the common geometric problems but also the optimization ones, containing engineering constraints. For example, one can find such a configuration of a complex sketch than the sum of areas (engineering constraints) of some closed contours consists exactly N percents of the area of the whole sketch.

The first step of validation of the modeler has been achieved with success for a set of examples in 2D-geometry.

3 Constraint Solver in Conceptual and Assembly Design

A CAD/CAM products mostly consist of a number of parts (which are made of features) that are connected to each other. The ideal product modeling system should therefore be able to support the modeling of all parts and their connections. Assembly constraints provides information on which component is connected to which other component in what way (face-contact), thus representing a model of an assembly. A CAD/CAM system must maintain the consistency of the designed product.

The design of a product can be thought of as a top-down (Conceptual Design) and/or bottom-up (Assembly Design) processes. Both of them can be considered as a sequence of phases, each of which adds information to the product model.

In the early phases of conceptual design, in which all global product requirements are gathered into the model, the designer does not yet want to think about all kinds of details that are not directly related to these requirements. In these phases, the designer only wants to specify those parts and constraints that are needed to satisfy the global requirements.

An assembly is a collection of parts, assembly features (coordinate systems, datum entities) other assemblies, and assembly properties. In the assembly the designer takes the ready-to-use parts and connects them by constraints according to the product requirements. If changes occur in one component the constraints can take care of the propagation of these constraints. This propagation can be authomatically done by solvers like *NeMo+*. Moreover, in the case when there exists libraries, catalogues of standard elements (parts, products), *NeMo+* can be used for the intelligent search of such elements. The *NeMo+* object-oriented language allows the designer to specify the query in high-level terms of the given data domain. It is possible to associate to the query more sofisticated requirements such as systems of equations, inequalities, rules (conditions), diagrammes, indicate the possible alternatives, etc. The end-user query is associated to the *NeMo+* model, elaborated and implemented by an expert. The solver, as we have mentioned before, returns the subdefinite result (the set of possible solutions) and one exact solution.

For example, a fragment of an expert model, which provides the choice of a bearing type from the given catalog, looks as follows:

```
B:Bearing;
B.P0 == B.Fr + (0.5 *B.Fa);  B.L10=(B.C / B.P)^B.p;
if B.Types2 == 2 then
 B.d == [ 3.0, 160.0 ];
 B.D == [ 10.0, 240.0 ];
 if ((311 <== B.Num)/\(B.Num <== 486)) // Kind of joints
 then ((-30 <== B.T)/\(B.T <== 110));  // Limits of temperature
 end;
 if (((B.T>>20) /\ (B.T<<120) /\ (B.nu>>5.01) /\ (B.nu<<400))
   \/((B.T>>20) /\ (B.T<<120) /\ (B.nu1>>5.0) /\ (B.nu1<<400)))
 then DIAGRAMME2 ( B.nu, B.T, B.nu1 );
 end;
end;
```

It should be noted once more that the values of parameters in all assembly or conceptual design components can be subdefinite. They become more and more exactly in the design process, when new components and new constraints arise in the product. This is the main difference between the proposed approach and the well-known existing industrial CAD/CAM systems, which assume that components are completely specified before assembly modelling is performed.

4 Constraint Solver in Knowledge Component

The use of the Knowledge component in a CAD system gives to the designer the following possibilities:

- Create and manage rules and knowledge bases;
- Check rules and knowledge base compliancy after design;
- Invoke knowledge base advisor during design.

The *NeMo+* environment can be used in the Knowledge component as a basic solver, which provides the rules checking, tables computation, optimization, constraint satisfaction, and solving a complex system of equations, inequalities, including real numbers, integers, strings, booleans, sets, and user-defined types.

Currently *NeMo+* is incorporated in the Knowledge component of a CAD/CAM system, where it represents a very clear concept for the user: a set of equations and inequalities. Such a set is defined in terms of mathematical equalities and inequalities and can include arithmetic, trigonometric and other standard mathematical functions. The user can arbitrary divide the parameters included in the set of equations (e.g. cost, material, distance, angle, area, volume, etc.) into two groups: inputs and outputs. Values of the input parameters are taken from the product, the output ones should be calculated by the solver. So, the interactive changing of input values enforces outputs to be recalculated by *NeMo+*. This behavior allows the user to optimize any parameter under design by easy switch between inputs and outputs. The implementation of the set of equations has been done by D.M.Ushakov.

5 Constraint Solver in Manufacturing & Data Management

Increasingly, CAD/CAM research is concerned with developing an integrated approach, incorporating the activities of design, manufacturing, process management and maintenance.

An advanced CAD/CAM system will have the following capabilities:

1) Integrate design-to-order and scheduling so as to calculate delivery dates.

2) Allocate resources and schedule the work of different teams, throughout the product life cycle, from design, through production and to disposal.

Obviously, the advantages of *NeMo+* for solving a calendar planning and job-shop scheduling problem is the possibility to deal with intervals of beginning, ending, and duration of jobs [13]. In order to solve more efficiently time scheduling problems, a specialized library of *NeMo+*, a JobShopScheduler, was implemented.

JobShopScheduler is a solver for use by applications with a need for solving job-shop scheduling problems such as well-known bridge building planning problem. This solver deals with jobs (that may, in turn, consist of smaller sub-jobs), which need to be scheduled according to constraints that link those jobs together. The constraints may concern jobs' precedence, their possibilities to perform at a given time, simultaneously with another jobs etc. The important feature of the solver is the presence of the notions of a resource, resource pool, and resource allocation. This allows us to state and solve complex optimization problems where jobs' processing requires certain resources and resources have limited capacity.

One of the possibilities of the problem specification as PERT chart is shown in Fig.2.

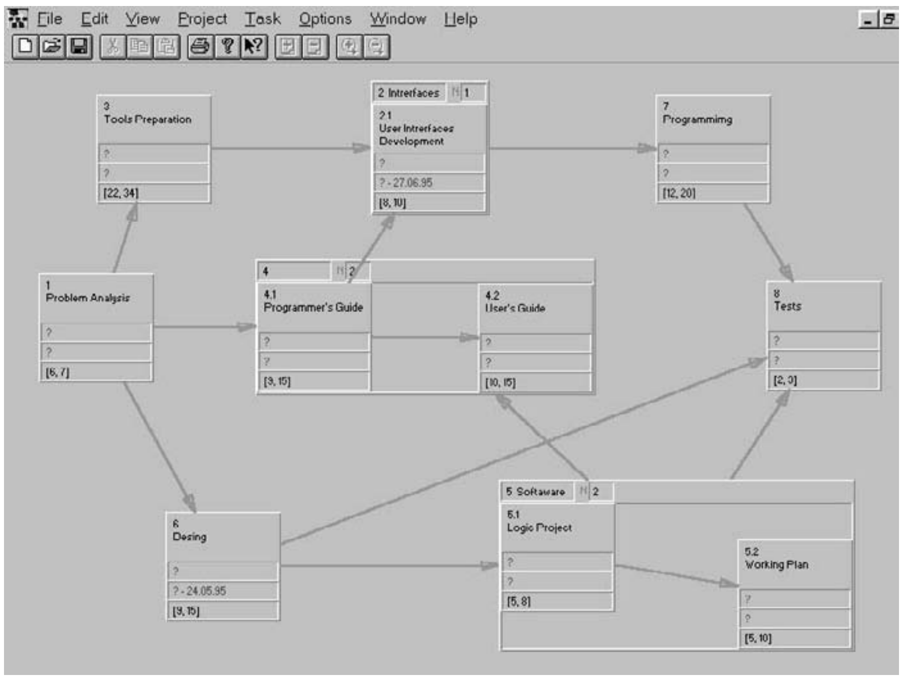


Fig. 2. Specification of a job-shop scheduling problem

JobShopScheduler is implemented as a C++ library which includes a set of basic constraint types derived from *NeMo+* ones, and also high-level constraints expressing most often used relationships between jobs and resources. This library has been implemented by V. S. Markin.

6 Conclusion

In the paper, we proposed the way to use constraint programming solvers in different components of a CAD system. In order to allow end-users to make a maximum profit, it is necessary that features support approximately known values of designed entities, and these values should be persistent in the model. The *NeMo+* constraint programming environment (or the solver with the same capabilities) is proposed to be used.

NeMo+ is implemented in C++ under *Windows* and UNIX platforms. There are no restrictions on the kind of constraints it can solve. One can build *NeMo+* specialized solvers in order to make it more efficient.

The applicability of this approach was proven by prototyping the geometric, and JobShopScheduler solvers in a CAD/CAM programming development environment, and by an integration of the *NeMo+* solver in the Knowledge component of the CAD/CAM system.

The forthcoming work will include the extention of *NeMo+* possibilities to process not only the functions implemented in its libraries, but also the external functions. Another interesting topic for *NeMo+* is the distributed and collaborative design. We hope, that in the future, *NeMo+* (or a *NeMo+*-like sover) will be integrated in the kernel of a CAD/CAM system for providing a general-purpose solution for the update engine of a CAD/CAM system.

References

1. Montanari, U. *Networks of Constraints: Fundamental Properties and Application to Picture Processing*, Information Science, V. **7**, (1974), P. 95–132.
2. Shvetsov I., Telerman V., Ushakov D. *NeMo+: Object-Oriented Constraint Programming Environment Based on Subdefinite Models*, LNCS **1330** (1997).
3. Telerman V., Ushakov D., Sidorov V. *Object-Oriented Constraint Programming Environment NeMo+ and its Applications*, Proc. of the 9th Internat. Conf. on Tools with Artificial Intelligence, ICTAI'97. — IEEE Computer Society, Newport Beach, California, USA, (1997).
4. Narin'yani, A.S. *Subdefiniteness and Basic Means of Knowledge Representation*, Computers and Artificial Intelligence, Bratislawa, **2**, No.5, (1983), 443–452.
5. Telerman V., Ushakov D. *Subdefinite Models as a Variety of Constraint Programming*, Proc. of the 8th Internat. Conf. on Tools with Artificial Intelligence, ICTAI'96. — IEEE Computer Society, Toulouse, France, (1996), 157–163.
6. Ushakov D. *Some Formal Aspects of Subdefinite Models.* Preprint No. 49, A. P. Ershov Institute of Informatics Systems, Siberian Division of Russian Academy of Sciences, Novosibirsk, (1998), 23 p.
7. Telerman V., Ushakov D. *Data Types in Subdefinite Models*, Lect. Notes in Comp. Sci. **1138** (1996), pp. 305–319.
8. Telerman V., Sidorov V., Ushakov D. *Interval and Multi-interval Extensions in Subdefinite Models*, Intern. Conf. on Interval Methods and Computer Aided Proofs in Science and Engineering (INTERVAL'96), Wurzburg, (1996), 131–132.
9. Lipski S., Sidorov V., Telerman V., Ushakov D. *Database Processing in Constraint Programming Paradigm Based on Subdefinite Models*, Joint NCC and IIS Bulletin, Comp.Science, No. 12, Novosibirsk, (1999), 29–31.

10. *The 2D DCM Manual (Version 3.7.0)*, D-Cubed Ltd., July, 1999, 314p.
11. Brown Associates Inc. *Applicon's GCE: a strong technical framework*, June 1993.
12. Thornton A.C. *Constraint specification and satisfaction in embodiment design*, PhD thesis, University of Cambridge, 1993.
13. Narin'yani A.S., Borde S.B., Ivanov D.A. *Subdefinite Mathematics and Novel Scheduling Technology*, Artificial Intelligence in Engineering, **11**, (1997).

A Graphical Interface for Solver Cooperations

Laurent Granvilliers and Eric Monfroy

IRIN
University of Nantes
F-44322 Nantes cedex 3, France
{granvilliers,monfroy}@irin.univ-nantes.fr

Abstract. We propose a language composed of basic graphical components. By assembling these components as in a Lego game, solver cooperations can be visualized. The advantage is to represent by simple figures complex cooperations that usually requires tedious descriptions. We illustrate our language by implementing some well-known cooperative solvers.

1 Introduction

Solver cooperation [9] is now well-known as a concept for improving efficiency and performance of constraint solvers. Generic solvers are generally far too inefficient for solving numerous real-life problems. However, a large part of these problems can often be handled by "incomplete" but specific and efficient solvers; furthermore, a solver can pre-process constraints in order to ease and speed-up a second solver.

The most usual type of cooperations (that we call *ad-hoc* cooperations) are based on one cooperation concept (*e.g.*, sequential solving process such as in **CoSAc** [14] or in the system of Beringer and Debacker [4], or concurrent communication such as in the system of Marti-Rueher [11]) and one solving strategy: the solvers are known, and the rooting of constraints through the solvers is fixed *a priori*. Examples of such cooperations are [4,11,14]. Implementing *ad-hoc* cooperations is a tedious task that involves several different problems, such as implementing communication between solvers, fixing interoperability problems, filtering constraints, synchronizing solving processes, and in the worst case re-implementing solvers from scratch.

On the one hand, cooperation languages [12,17,8,7] recently emerged as a new concept for designing and automatically implementing (such as in [12]) solver cooperations as expressions of a calculus-like language. However, cooperation expressions quickly become difficult to read. Moreover, interactions and communications between solvers are not explicit, but are hidden in the definitions of the primitives for building expressions.

On the other hand, the concept of coordinating a number of activities, running concurrently in a parallel and distributed fashion, has recently received wide attention (*e.g.*, see [16]). Visual interfaces to such languages already exist, such as the Visifold interface [5] for the control-driven coordination language **Manifold** [1].

Some works have already been conducted to relate cooperation and coordination [13], and to use coordination facilities for solver cooperation [2]. However, to our knowledge, visual interfaces (similar to visual interfaces of coodination languages) have not been integrated in solver cooperation languages.

In this paper, we propose a language for graphically designing solver cooperations. This language is composed of some few basic components from which more complex bricks and solver cooperations are built such as in a Lego game. Usual patterns of cooperation (such as sequential, concurrent, and parallel solving processes), and standard control on constraint routing (such as conditional, fixed-point, selections) can easily be designed linking solver, control, filter, and selection agents with channels of communication. Complex cooperations are then built connecting these patterns of constraint processing. This language aims at representing graphically in a unified and simple way solver cooperations. The growing capacity of this language is tremendous: first, patterns of cooperations can become new bricks of the language, and second, new basic components can easily be integrated. Moreover, adding a new component correspond to implementing a new module that does not interfere but interact with previous pieces of code.

We illustrate the practicality of our language by "simply" visualizing some *ad-hoc* cooperations (such as the ones of [4,3]) that normally require long descriptions.

The outline of this paper is the following: definitions for constraints, solvers and filters are presented in Section 2. Basic graphical components are described in Section 3 in terms of communicating agents. Using these components, some standard patterns of cooperation are designed in Section 4, before visualizing some well-known ad-hoc cooperations in Section 5. We finally conclude in Section 6.

2 Framework

Let $\mathcal{D}$ be a set called the universe, $\mathcal{F}$ a set of function symbols, $\mathcal{R}$ a set of relation symbols, $\Sigma = (\mathcal{D}, \mathcal{F}, \mathcal{R})$ a structure, and $\mathcal{X} = \{x_1, \ldots, x_n\}$ a set of variables. A *constraint language* $\mathcal{L}$ is a non-empty set of first order $(\Sigma, \mathcal{X})$- formulae. Given a constraint c on variables $x_1, \ldots, x_n$, let ρ_c denote the underlying relation on $\mathcal{D}^n$. The relation ρ_c associated to the constraint c on $x_{i_1}, \ldots, x_{i_l}$ is extended to the set $\rho_c^+ = \{(v_1, \ldots, v_n) \in \mathcal{D}^n \mid (v_{i_1}, \ldots, v_{i_l}) \in \rho_c\}$. A *constraint store* C is given as a set $\{c_1, \ldots, c_m\}$ of constraints from $\mathcal{L}$ interpreted as the conjunction $c_1 \wedge \cdots \wedge c_m$. The solutions of C (denoted by $Sol(C)$) are defined as:

$$Sol(C) = \bigcap_{i=1}^{m} \rho_{c_i}^+$$

$\mathcal{L}_S$ represents the set of stores built upon the constraint language $\mathcal{L}$. We can now define the notion of solver in our scheme.

Definition 1 (Solver). *Consider a constraint language $\mathcal{L}$. Then, a solver S on $\mathcal{L}$ is a computable function $S: \mathcal{L}_S \longrightarrow \mathcal{L}_S$.*
A solver is said:

$$\begin{array}{ll} \textit{correct:} & \textit{if } \forall C \in \mathcal{L}_S,\ Sol(S(C)) \subseteq Sol(C) \\ \textit{complete:} & \textit{if } \forall C \in \mathcal{L}_S,\ Sol(C) \subseteq Sol(S(C)) \end{array}$$

With respect to Definition 1, Gröbner basis computation, Simplex, Gaussian elimination, factorization of polynomials, trigonometric transformations are solvers. No property is required *a priori* for solvers. However, some properties of solver cooperations are induced from solver properties (see *e.g.*, dispatcher in Section 4).

Definition 2 (Cooperation). *Consider k solvers $S_1, \ldots, S_k$ respectively on $\mathcal{L}_1, \ldots, \mathcal{L}_k$, and a constraint language $\mathcal{L}$. We say that solvers $S_1, \ldots, S_k$ on $\mathcal{L}_1, \ldots, \mathcal{L}_k$ can cooperate on $\mathcal{L}$ if:*

$$\forall i \in [1, k],\ \mathcal{L}_i \subseteq \mathcal{L}.$$

Stores from $\mathcal{L}_i$ are called admissible constraints of S_i on $\mathcal{L}$.

The role of filters is very important for cooperations on a language $\mathcal{L}$. They select parts of constraints stores, *i.e.*, subsets of stores in order to:

1. select constraint stores a solver S on $\mathcal{L}' \subseteq \mathcal{L}$ can actually handle, *i.e.*, the admissible constraints of S on $\mathcal{L}$,
2. and, treat efficiently subsets of stores by specific solvers.

Definition 3 (Filter). *Consider a constraint language $\mathcal{L}$. A filter on $\mathcal{L}$ is a computable function $\varphi: \mathcal{L}_S \longrightarrow \mathcal{L}_S$ such that:*

$$\forall C \in \mathcal{L}_S,\ \varphi(C) \subseteq C$$

Usual filters are $\varphi_{\text{eq\_poly}}$ to filter polynomial equations, φ_{lin} to filter linear constraints, *etc.*

Property 1. A filter is a complete and non correct solver.

Filters can be combined using intersection, union, and complementary operators to compose more complex filters. The results are the expected standard set operators on stores of constraints. Consider two filters φ_1 and φ_2 on $\mathcal{L}$. Then, for all $C \in \mathcal{L}_S$, we have:

$$\begin{array}{l} (\varphi_1 \cup \varphi_2)(C) = \varphi_1(C) \cup \varphi_2(C) \\ (\varphi_1 \cap \varphi_2)(C) = \varphi_1(C) \cap \varphi_2(C) \\ \bar{\varphi}(C) = C \setminus \varphi(C) \end{array}$$

The reader can refer to [7] for more complex examples and a more complete presentation of filters.

3 Basic Graphical Components

We design a set of graphical —basic— components that can be combined to implement solver cooperations. The underlying model is based on *agents* — solvers, filters, selectors, *etc.*— acting on constraints. An agent receives data on input *ports*, transforms them, and puts the resulting data on output ports. Ports of agents are connected by *channels*, where a channel transfers a constraint store from an output port of an agent to an input of another agent.

The basic graphical components are presented in Figure 1. Their precise meanings will be explained in the following.

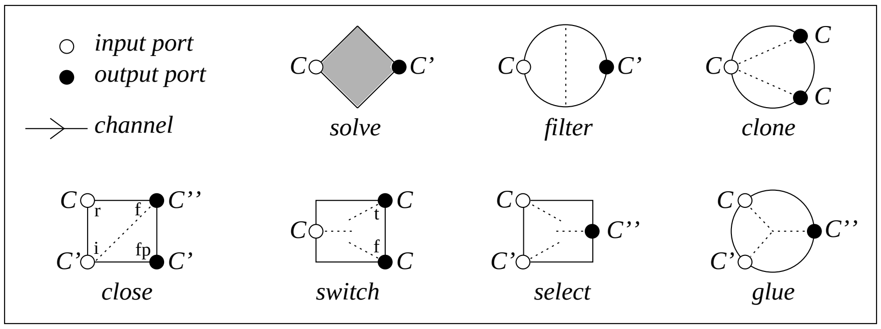


Fig. 1. Basic graphical components

3.1 Communication

Communications of constraint stores are modeled by ports — holes on agents — and channels — linkers of ports. A *port* either models an *input* of an agent, or an *output*. A *channel* implements a one-to-one (from an output port to an input port) communication of constraint stores.

In the following, we propose agents achieving solver computations, and filtering, controlling, and redirecting communications.

3.2 Solving Agents

Solving agents (Primitive *solve* in Figure 1) capture the computational part of cooperations. In fact, most of existing systems integrate a restricted set of algorithms (such as Gröbner bases [6], consistency techniques [10], interval methods [15], *etc.*) that are seen as black-box solvers. Consider a solver S. Then, from an input constraint store C (available on the input port), if C belongs to the constraint language of S, then S is applied on C ($S(C) = C'$), and the new store C' is delivered on the output port; otherwise, S is not applied, and $C' = C$:

```
when $C$ on input
    if $C$ in language of $S$
        then put $S(C)$ on output
        else put $C$ on output
```

3.3 Transformer Agents

Transformer agents are essentially solvers processing constraints by means of set operations: restriction, union, conjunction, *etc.* We briefly discuss four kinds of useful operations.

Filter. A *filter agent* (Primitive *filter* in Figure 1) applies a filter φ (see Section 2) on C to extract a subset of the constraints verifying some property, *i.e.*, $C' = \varphi(C)$ such that $C' \subseteq C$:

```
when $C$ on input,  put $\varphi(C)$ on output
```

A filter is generally used to preprocess a constraint store before applying a specific solver.

Clone. Cloning (Primitive *clone* in Figure 1) a constraint store C consists in duplicating C on every output ports.

```
when $C$ on input,  put $C$ on output 1  and  put $C$ on output 2
```

Note that the combination of several cloning agents increases the number of clones: $n-1$ combined cloning agents lead to n clones. Cloning agents are useful for realizing different usual tasks of cooperation, such as modeling concurrent algorithms, and processing of sub-stores.

Glue. Gluing (Primitive *glue* in Figure 1) constraint stores C and C' means generating their union $C'' = C \cup C'$. This operation is performed when all input stores are available on input ports.

```
when $C$ on input 1, and $C'$ on input 2,  put $C \cup C'$ on output
```

Typically, it gathers together results generated by cooperative solvers acting on the same constraint store (competitive concurrent solvers), or acting on disjoint sub-stores (cooperative concurrent solvers).

Select. The *selection* (Primitive *select* in Figure 1) of two input constraint stores C and C' transfers just one of them to the output port. We have either $C'' = C$ or $C'' = C'$:

```
when $C$ on input 1 and $C'$ on input 2,  put $C$ or $C'$ on output
```

3.4 Control Agents

A control agent manages the rooting of constraint stores during the solving process. We identify agents modeling switches and fixed-point computations.

Switch. A *switch* (Primitive *switch* in Figure 1) is based on a P function from stores to Booleans: P checks whether input constraint stores verify a given property or not. Consider a P function and an input store C: if $P(C)$ is true, then C is transferred to the output port t, otherwise to the output port f:

```
when $C$ on input,
    if $P(C)$
          then put $C$ on $t$
          else put $C$ on $f$
```

Switches represents conditional of [12]: they are very important to dynamically control solver cooperations.

Close. A more complex kind of switch (Primitive *close* in Figure 1) is devoted to *fixed-point* computations. It is based on the detection of equivalent consecutive input constraint stores. For this purpose, such an agent has a memory —a constraint can be stored between consecutive applications—, two input ports i (input) and r (re-enter), and two output ports f (follow) and fp (fixed-point). The computation processes are as follows:

– Initially, the memory is empty. The first time a constraint store C is received on i (input port), it is stored in the memory, and also put on port f (follow port).
– Then, when a constraint store C' arrives on r (re-enter port), it is compared with the memory. If they are equivalent, C' is put on port fp (fixed-point port) and the memory is cleared and reset for next use —the fixed-point is just detected.
– If they are different, C' is put on port f (follow port) and the memory is updated with C'.

The close agent can be described as follows:

```
when $C$ on $i$, put $C$ on $f$ and $M = C$
when $C$ on $r$,
     if $C = M$
         then put $C$ on $fp$
         else put $C$ on $f$ and $M = C$
```

4 Standard Patterns of Cooperation

We now describe two patterns involved in numerous cooperations.

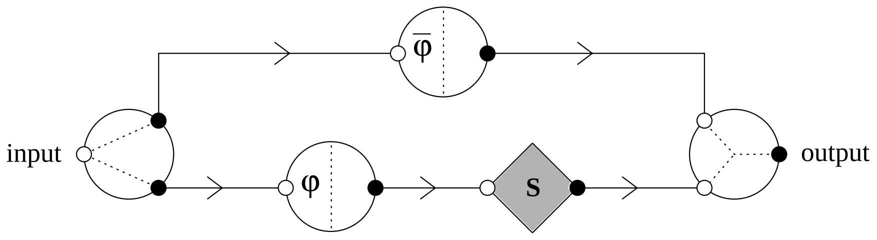


Fig. 2. Solver protection

Solver protection. When using a *solver protection* (Figure 2) solvers process only subsets of the constraint store (i.e., their admissible constraints, constraints they can effectively handle), while the rest of the input store is preserved, and used to create the new constraint store. More precisely, the input store is first cloned. Then,

– on one branch, the store is filtered by φ to extract from a constraint store C the admissible constraints of S. S is then applied on the resulting store;
– on the other branch, the filter $\bar{\varphi}$ (*i.e.*, the complementary of φ) filters $C \setminus \varphi(C)$.

The output of S, and the non-admissible constraints of S are then glued together: the final output of the protection is: $(C \setminus \varphi(C)) \cup S(\varphi(C))$.

Property 2. Consider a solver S, and φ to filter its admissible constraints. If S is complete, then the protection of S is also complete.

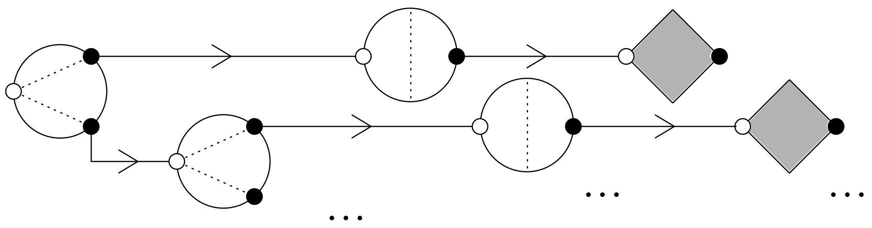


Fig. 3. Dispatcher of a store to solvers

Dispatcher. A *dispatcher* (Figure 3) distributes constraint store to n solvers (each of them being associated with a specific filter) using $n-1$ cloning agents. Note that it can also be combined with a solver protection to preserve the whole of the input store.

5 Modeling Existing Systems

We design three existing cooperative solvers to illustrate the feasibility of our approach.

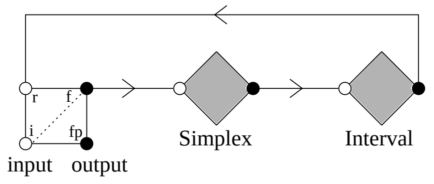


Fig. 4. Cooperation Simplex-interval solver on linear constraints

Cooperation for linear constraints. The system of Beringer and Debacker [4] is devoted to linear constraints: domains of variables are reduced with an interval solver while a Simplex-like solver tries to detect inconsistency of stores and to fix variables. As soon as new information is deduced by one of the solvers, it is communicated to the other one. The process terminates when a fixed-point is reached, *i.e.*, none of the solver is able to deduce new facts anymore. This cooperation is visualized in Figure 4. The solvers are applied in sequence, each one processing the whole constraint store. The detection of a fixed-point is realized by a closure agent.

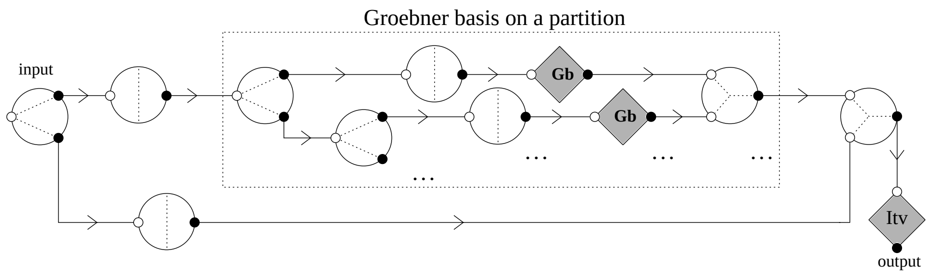


Fig. 5. Cooperation Gröbner basis-interval solver on arbitrary constraints

Gröbner basis computation and interval solver. The cooperation of Gröbner basis computation used as a preprocessing for an interval solver is visualized in Figure 5. The input constraint store is filtered in order to extract the set of polynomial equations. A set of Gröbner bases is then computed for a partition of the set of polynomial equations (construction cloning-filter-solver). The input of the interval solver is the union of the computed Gröbner bases and the input constraints that are not polynomial equations.

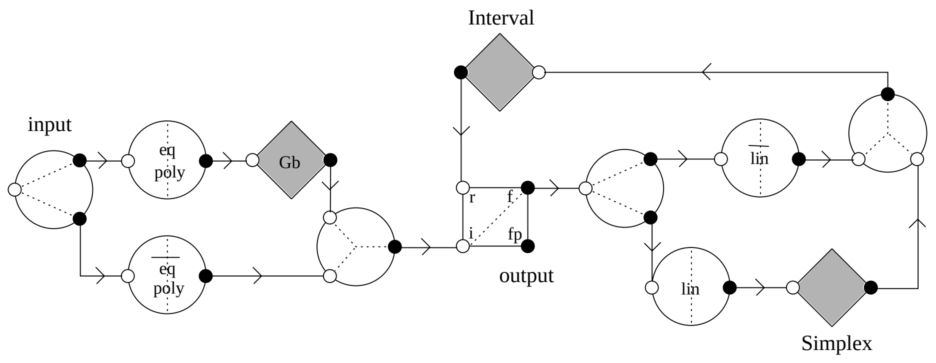


Fig. 6. Cooperation Gröbner basis-Simplex-interval solver

Gröbner basis computation, Simplex, and interval solver. Figure 6 illustrates the following cooperation: a Gröbner basis is generated for polynomial equations, and they are combined with the other input constraints. Then, a fixed-point of the Simplex and an interval solver applied in sequence is computed. A filter of linear constraints is necessary in front of the Simplex, while the interval solver handles all constraints (linear and nonlinear constraints are combined after the application of Simplex).

6 Conclusion

In this paper, we have proposed a language for visualizing solver cooperations in a unified and graphical manner. This language is composed of basic components that are then linked together by channels in order to graphically represent solver cooperations. Complex cooperations that usually require long explanations are described by a simple figure in our language. The growing capacity of the language are tremendous since integrating a new basic component corresponds to adding a module in a component based-framework and does not provoke any side-effect.

We are confident in the practical realization of our language to automatically implement solver cooperations from their graphical descriptions: cooperation features are similar to primitives of **BALI** (which has already been implemented), and more complex visual interface (such as [5]) have already been realized for complete coordination languages (such as Manifold [1] which requires several complex communication and interaction features).

In the future, we plan to extend our language by introducing several types of communication in order:

1. to enable solvers waiting for complementary constraints,
2. and, to manage disjunctions of constraints (*e.g.*, several possible solutions) as different constraint stores requiring different solving processes.

References

1. F. Arbab. *Manifold2.0 reference manual*. CWI, Amsterdam, The Netherlands, May 1997.
2. F. Arbab and E. Monfroy. Coordination of Heterogeneous Distributed Cooperative Constraint Solving. *ACM SIGAPP Applied Computing Review*, 6:4–17, 1998.
3. F. Benhamou and L. Granvilliers. Automatic Generation of Numerical Redundancies for Non-Linear Constraint Solving. *Reliable Computing*, 3(3):335–344, 1997.
4. H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In *Logic programming: formal methods and practical applications*. Elsevier Science Publisher B.V., 1995.
5. P. Bouvry and F. Arbab. Visifold: A visual environment for a coordination language. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 403–406. Springer-Verlag, April 1996.
6. B. Buchberger. Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory*, pages 184–232. 1985.
7. C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proceedings of Andrei Ershov Third International Conference Perspective of System Informatics, PSI'99*, volume 1755 of *Lecture Notes in Computer Science*, pages 402–415, Novosibirsk, Akademgorodok, Russia, 2000. Springer-Verlag.
8. C. Castro and E. Monfroy. Basic Operators for Solving Constraints via Collaboration of Solvers. In J. A. Campbell and E. Roanes-Lozano, editors, *Proceedings of the 5th International Conference on Artificial Intelligence and Symbolic Computation (AISC'2000)*, volume 1930 of *Lecture Notes in Artificial Intelligence*, pages 142–146, Madrid, Spain, 2001. Springer.
9. L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-Interval Cooperation in Constraint Programming. In *Proceedings of the 26th International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 150–166, University of Western Ontario, London, Ontario, Canada, 2001. ACM Press.
10. A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
11. P. Marti and M. Rueher. A Distributed Cooperating Constraints Solving System. *International Journal on Artificial Intelligence Tools*, 4(1&2):93–113, 1995.
12. E. Monfroy. The Constraint Solver Collaboration Language of BALI. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 211–230. Research Studies Press/Wiley, 2000.
13. E. Monfroy and F. Arbab. *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter Constraints Solving as the Coordination of Inference Engines, pages 399–422. Springer-Verlag, Omicini, A. and Zambonelli, F. and Klusch, M. and Tolksdorf, R. edition, 2001.

14. E. Monfroy, M. Rusinowitch, and R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of ACM Symposium on Applied Computing (SAC'96), Philadelphia, PA, USA*, pages 63–72. ACM Press, February 1996.
15. R.E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.
16. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46: The Engineering of Large Systems, 1998.
17. A. Semenov, D. Petunin, and A. Kleymenov. GMACS: the general-purpose module architecture for building cooperative solvers. In *Proceedings of the ERCIM/CompulogNet Workshop on Constraint programming*, Padova, Italy, 2000.

Abstract Computability of Non-deterministic Programs over Various Data Structures

Nikolaj S. Nikitchenko

Department of the Theory of Programming
Faculty of Cybernetics, Kiev National University
Vladimirskaja 64, Kiev, 01033, Ukraine
nikit@unislav.kiev.ua

Abstract. Partial multi-valued functions represent semantics of non-deterministic programs. The notion of naturally computable partial multi-valued function is introduced and algebraic representations of complete classes of naturally computable functions over various data structures are constructed.

1 Introduction

The title of this paper is a clear reminiscence of A.P. Ershov's articles "Abstract computability on algebraic structures" [1] and "Computability in various domains and bases" [2]. This tight connection of titles is not accidental and has a long history. The story goes back into 1984 when the author being on a sabbatical leave from the Kiev State University spent half a year in Novosibirsk at the A.P. Ershov's department of the Computer Center of the Siberian Branch of the Soviet Academy of Sciences. During his stay in Novosibirsk the author had the possibility to study the intentions of Ershov's works on computability, was fascinated by his ideas and tried to follow them in his own investigations. The author is grateful to A.P. Ershov for support in his work on the topic.

The main objective of Ershov's works on computability was the necessity to develop for computer sciences their own fundamental conceptions of the computability theory [1]. Such a theory must define computability for various subject domains and different systems of basic operations; clearly distinguish combinatorial and "executable" aspects of computability; be independent of specific program syntax and mechanisms of program evaluation [2].

The author's research on computability are based on the following ideas of A. P. Ershov:

– the notion of abstract computability must be oriented on abstract models of programs,

– abstract computability has a relative character,

– the notion of determinant[1] can be used for the definition of function computability.

[1] A.P. Ershov understood determinants as sets of special terms constructed over given algebraic system [2].

To realise these ideas we

– construct abstract but powerful models of programs,

– define exact definitions of computability, which satisfy the described requirements,

– study the properties of introduced notions of computability.

Such definitions were first developed for the compositional model of programs [3,4]. The notions of natural and determinant computability were introduced and the complete classes of functions and compositions over different classes of named data were described. In this paper we extend this approach to new more general classes of program models, based on composition nominative principles [5]. The main extensions concern computability over nominative data for non-deterministic programs.

The paper is structured in the following way:

– first, we define composition nominative systems, which can be considered as abstract powerful program models,

– then, we discuss questions of computability in programming languages and define the notion of natural computability of partial multi-valued functions, which represent semantics of non-deterministic programs,

– at last, complete classes of computable partial multi-valued functions over different specializations of nominative data structures are described.

2 Composition Nominative Approach to Program Definition

The main goal of the approach is to construct a clear hierarchy of adequate program models of various levels of abstraction and generality. Dialectical logic developed by G.W.F. Hegel and his followers is used as a gnoseological (epistemological) foundation of this approach.

The approach is based on the following principles, which specify the main program notions.

Development principle (rising from abstract to concrete): the notion of program should be introduced as a process of its development, which starts from abstract understanding capturing essential program properties and proceeds to more and more concrete considerations thus gradually revealing the notion of program in its richness.

Applicativity (functionality) principle: at the highest abstraction level programs can be considered as functions which being applied to input data can produce output data.

Function nominativity principle: programs can be presented as names denoting functions which being applied to input data can produce output data.

Compositionality principle (V. Red'ko [6]): programs can be considered as functions which map input data into output data, and which are constructed from simpler programs (functions) with the help of special operations, called compositions.

Descriptivity principle: programs can be considered as descriptions (complex names) which denote functions constructed from simpler functions with the help of compositions.

These principles introduce five notions: data, function, function name, composition and description, which form the pentad of main program notions. Formalizations of such notions are usually based on the notion of set, thus giving set-theoretic formalizations of programs. Still, there are proposals to use instead the notion of function [7]. Here we will follow this way considering a function-theoretic approach.

Principle of function-theoretic formalization: program notions are formalized on a base of a function-theoretic approach.

Please note that we do not reduce the notion of set to the notion of function. But we do not either adopt the traditional reduction of the notion of function to the notion of set. Thus, we treat the both notions as mutually dependent on each other.

Taking into account the development principle we come to the conclusion that we need a special theory which permits to study functions on various abstraction levels. We propose to call such a theory as DEFT Theory (DEveloping FuncTion Theory). Abstract computability, presented in this paper, may be regarded as an integral part of this theory.

Functions, which maps elements of A into R, are considered in the most general way as partial multi-valued functions. In this case functions are not uniquely represented by their graphs, therefore we will additionally take into consideration the sets of elements on which functions can be undefined (undefinedeness sets). For example, function $f = [1 \mapsto 1, 1 \mapsto 2, 1 \mapsto, 2 \mapsto, 3 \mapsto 1]$ has a binary relation $\{(1,1), (1,2), (3,1)\}$ as its graph and a set $\{1,2\}$ as its undefinedeness set. We will use the following notations for the classes of functions:

– $D \overset{m}{\to} D$ – partial multi-valued functions,

– $D \to D$ – partial single-valued functions,

– $D \overset{t}{\to} D$ – total single-valued functions.

Program models on high abstraction levels can be presented as composition nominative systems [5]. Such a system may be considered as a triple of the following simpler systems: composition, nominative, and denotational systems. Composition system defines semantic aspects of programs, nominative system defines program descriptions (syntactic aspects), and denotational system specifies meanings of descriptions. Here we will consider only composition systems which are triples of the form $< D, F, C >$, where D is a set of data, on which programs are defined, $F \subseteq (D \overset{m}{\to} D)$ is a class of partial multi-valued functions, representing program semantics, and C is a class of compositions over F, representing program construction means.

These definitions are specialised for more concrete levels. We can distinguish three main levels: abstract, Boolean, and nominative levels [5]. The last level is the most interesting level for programming. On this level program data are considered as nominative data, which are constructed hierarchically with the help of naming relations.

For given sets of names V and basic elements W the class of nominative data $ND(V,W)$ can be presented by induction or by the following recursive definition:

$$ND(V,W) = W \cup (V \overset{m}{\to} ND(V,W)).$$

Main unary operations over nominative data with the name v as a parameter are the following.

- Naming operation $\Rightarrow v$. Being applied to some data d it yields the nominative data having the only component with the name v and the value d.
- Partial multi-valued denaming operation $v\Rightarrow$. Being applied to some nominative data d it yields one (arbitrary chosen) value of the name v, if at least one component with the name v is in d.
- Deleting operation $\backslash v$. Being applied to some nominative data d it deletes one (arbitrary chosen) component with the name v, if such a component is in d.
- Checking operation $v!$. Being applied to some data d it yields the empty nominative data $\emptyset$, if v has at least one value in d, and d, if v has no value in d.

Main binary operations over nominative data are the following.

- Override operation [8] $\dagger$. Being applied to nominative data d and d' it yields a new nominative data, which has as its components all components of d' and those components of d, the names of which do not occur in d'.
- Union operation $\cup$ combines two nominative data yielding a new data, which has all components of the arguments.
- Subtraction operation $\backslash$ deletes from the first nominative data those components, which belong to the second nominative data.

We also use non-deterministic *choice* operation, which on d yields d or $\emptyset$.

Functions of the set $F = ND(V,W) \overset{m}{\to} ND(V,W)$ are called nominative functions. The main compositions defined over F are binary compositions, described by the following formulas, where $f, g \in F, d \in ND(V,W)$.

- Multiplication (functional composition): $(f \circ g)(d) = g(f(d))$.
  Note that f is applied first and g second.
- Iteration. This composition is similar to the operation *while_do* and can be easily defined inductive, but for simplicity we present here its recursive definition:

$$(f * g)(d) = \begin{cases} d, & \text{if } f(d) = \emptyset, \\ (f * g)(g(d)), & \text{if } f(d) \neq \emptyset. \end{cases}$$

  Here $\emptyset$ is the empty nominative data.
- Overriding: $(f \nabla g)(d) = f(d) \dagger g(d)$.
- Summation: $(f \sqcup g)(d) = f(d) \cup g(d)$.

In all cases, when certain application of a function to some data in the right-hand side of a formula is not defined, then the result of the left-hand side of the formula is also undefined.

Concretizations of nominative data can represent various data structures, such as records, arrays, sets, tables, etc. [4,5]. For example, a set $\{s_1, s_2, ..., s_n\}$ can be presented as a nominative data $[1 \mapsto s_1, 1 \mapsto s_2, ..., 1 \mapsto s_n]$, where 1 is treated as a standard name. Thus, we can formulate the following principle.

Data nominativity principle: program data structures can be presented as concretizations of nominative data.

Having defined composition nominative systems as powerful program models (models of programming languages), we can now specify special computability for such models.

3 Computability in Programming Languages

Conventional programming languages are usually called universal languages. It means that their programs define computable functions, and vice versa, any computable function may be represented by a certain program, written in such a language. But more thorough investigation reveals a number of difficulties, which are concerned with our understanding of computability in programming languages. Usually computability is understood as computability of n-ary functions defined on integers or strings. Such computability may be called Turing computability. But programming languages also work with other data structures and it turns out that for these structures programming languages, which are considered as universal, may not be universal. That is: their programs cannot represent all computable functions definable on these structures [5].

To study this problem we need its exact formulation. It seems natural to start with the set D of all data definable in a language and then consider this language as universal (or computationally complete), if its programs represent the class of all computable functions from D to D. Under this formulation it is well known that Pascal, for example, is not complete, because it is forbidden to use dynamic arrays and consequently it is impossible to write a program (procedure) for multiplication of matrixes with changing dimensions.

Typing is not the only reason of computational incompleteness. For conventional programming languages a more important reason is absence of facilities for dynamic construction and analysis of data structures. For example, conventional languages usually do not have facilities to check whether a variable has a value or not. Some languages, say, Ada, to cope with such difficulties use a mechanism of exceptions, like NO_VALUE_ERROR, INDEX_ERROR, RANGE_ERROR, etc. However, not much attention is paid to completeness problems in programming languages. Similar situation arises in database query languages, when many of them are computationally incomplete. The requirement of computational completeness was even declared in "The object-oriented database manifesto" [9].

So, the computational completeness of programming languages is not a trivial problem and calls for specific further investigations.

The completeness problem is not the only aim of our investigation. Now it is a common opinion that programs should be developed successively from abstract specifications via more concrete representations up to detailed implementations in chosen programming languages. And it is very important to connect completeness and computability problems with stages of program development. We intend to introduce such unified notions of computability and completeness that can be applied to every stage of program development and can be easily transformed when moving from stage to stage of development. Such a kind of computability should be applicable to data structures of different abstraction levels and is called abstract computability [1]. In fact, such computability is a relative computability – relative to data structures and operations over them.

Another facet of the problem is formulation of simple and clear descriptions of complete classes of computable functions on each level of abstraction. We shall construct algebraic representations of such classes. It means, that a complete class will be described as the closure of some basic functions under a certain (and very simple) class of compositions (operators over functions).

Many results are currently available in this area. We only mention [1,10,11,12, 13,14,15,16]. However, despite the richness of the available results, the attempt to apply them to our problems runs into various difficulties.

The point is that many approaches postulate certain requirements which, first, are far from obvious and themselves require substantiation, e.g., the existence of a universal computable function, as assumed by Strong, Freedman, Moschovakis, and second, are often inapplicable to specific data structures, e.g., the requirement of data enumerability of Mal'tsev's enumeration approach, or the requirement of Goedelisation of Wagner's approach (see refs. in [2]). Recall that we are concerned with computability defined in terms of data structures of programming languages. We will therefore try to motivate the proposed formalization of computability by using weaker postulates, from which other postulates may be obtained as corollaries (as suggested by Gandy [10] and Scott [11]). In other words, we will attempt to identify the key ideas of abstract computability, which can be combined to obtain concrete results.

We will study computational completeness of the classes of functions over nominative data. The difficulty of the problem lies in the fact that the notion "computability over complex data structures" by itself must first be defined and then, only on this basis, complete classes of functions can be described.

Here we restrict our consideration only by computability over finite data structures, which is called natural computability.

4 Natural Computability of Partial Multi-valued Functions

In order to formalise computability of functions over finite data structures, we first need to define such data. This is a difficult question, and we will accordingly adopt the following strategy: we will first define a special form of finite data structure and subsequently reduce data of other forms to this special form.

Our intuitive notion of a finite data structure is the following: any such datum d consists of several basic (atomic) components $b_1, ..., b_m$, organised (connected) in a certain way. If there are enumerably many different forms of organisation for finite data structure, each of these data can be represented in the (possibly non-unique) form $(k, < b_1, ..., b_m >)$, where k is the **datum code** and the sequence $< b_1, ..., b_m >$ is the **datum base**. Data of this form are called **natural data** [3]. More precisely, if B is any set and Nat is the set of natural numbers, then the set of natural data over B is the set $Nat(B) = Nat \times B^*$.

A set D is called a set of **finite data structure** (over B with respect to nat), if a set B and a total multi-valued injective[2] mapping $nat : D \overset{m}{\to} Nat(B)$ are given. This mapping nat is called the **naturalization** mapping, and the partial single-valued inverse mapping $nat^{-1} : Nat(B) \to D$, denoted by $denat$, is called the **denaturalization** mapping.

Very often the denaturalization mapping is called the abstraction mapping. We prefer to start with naturalization mapping as primary, because our definitions are developed in the direction from abstract levels to concrete ones.

The introduction of natural data and naturalization mappings enables us to reduce computability over D to special computability over $Nat(B)$, which is called code computability. To define this type of computability we should recall that in natural data the code collects all known information about datum components. Thus, code computability should be independent of any specific processing tools of the elements of the set B and can use only those tools which are independent of B and are explicitly exposed in natural data. The only explicit information in natural data is the datum code and the length of the datum base. Therefore in code computability the datum code plays a major role, while the elements of the datum base are "extras" that virtually do not affect the computations. The elements of a datum base may be only used to form the base of the resulting datum. To describe the code of the resulting datum and the order in which elements of the initial base are put into the base of the resulting datum, a special function of type $Nat^2 \overset{m}{\to} Nat \times Nat^*$ should be used. Such functions are called **index computable** functions. These considerations lead to the following definition.

A function $g : Nat(B) \overset{m}{\to} Nat(B)$ is called **code computable**, if there exists an index computable multi-valued function $h : Nat^2 \overset{m}{\to} Nat \times Nat^*$ such that for any $k, m \in Nat$, $b_1, ..., b_m \in B$, $m \geq 0$ $\quad g(k, < b_1, ..., b_m >) = (k', < b_{i_1}, ..., b_{i_l} >)$, if and only if $h(k, m) = (k', < i_1, ..., i_l >)$, $1 \leq i_1 \leq m, ..., 1 \leq i_l \leq m$. If at least one of the indexes $i_1, ..., i_l$ lies outside the interval $[1,\ m]$, or $h(k, m)$ is undefined, then $g(k, < b_1, ..., b_m >)$ is also undefined.

In other words, in order to compute g on $(k, < b_1, ..., b_m >)$, we have to compute h on (k, m), generate a certain value $(k', < i_1, ..., i_l >)$, and then try

[2] A multi-valued function is injective, if it yields different values on different arguments.

to form the value of the function g by selecting the components of the sequence $< b_1, ..., b_m >$ pointed to by the indexes $i_1, ..., i_l$.

It is clear, that index computability of $h : Nat^2 \overset{m}{\to} Nat \times Nat^*$ may be reduced by traditional methods of the recursion theory to computability of a certain (partial recursive) function $r : Nat \overset{m}{\to} Nat$.

We are ready now to give the main definition of this section.

A function $f : D \overset{m}{\to} D$ is called **naturally computable** (with respect to given B and nat), if there is a code computable function $g : Nat(B) \overset{m}{\to} Nat(B)$ such that $f = nat \circ g \circ denat$. The class of all naturally computable functions is denoted by $NatComp(D, B, nat)$.

The relations between introduced notions of computable functions can be presented with the help of the following diagram of natural computability, in which double-headed arrows denote multi-valued functions.

$$\begin{tikzcd}
d : D \arrow[r, twoheadrightarrow, "f"] \arrow[d, twoheadrightarrow, "nat"'] & d' : D \\
(k, < b_1, ..., b_m >) : Nat \times B^* \arrow[r, twoheadrightarrow, "g"] \arrow[d] \arrow[ur, phantom] \arrow[r, bend right, dash] & (k', < b_{i1}, ..., b_{il} >) : Nat \times B^* \arrow[u, "denat"'] \\
(k, m) : Nat \times Nat \arrow[r, twoheadrightarrow, "h"] \arrow[d] & (k', < i1, ..., il >) : Nat \times Nat^* \arrow[u] \\
n : Nat \arrow[r, twoheadrightarrow, "r"] & n' : Nat \arrow[u]
\end{tikzcd}$$

Analysing this diagram we can conclude, that natural computability as a generalization (relativization) of enumeration computability. In fact, for $B = \emptyset$ code computability reduces to partial recursive computability on Nat, and natural computability reduces to enumeration computability (with respect to nat). Natural computability may be also used to define computability of polymorphic functions. Therefore, the notions of code and natural computability defined above are quite rich.

We start investigation of natural computability with the following question: how strongly the class of naturally computable functions depends on specific features of the naturalization mapping?

It turns out, that the class $NatComp(D, B, nat)$ is stable under "effective" transformations of nat. Thus, the situation is the same as for enumeration computability. More formally, we say that a naturalization mapping $nat' : D \overset{m}{\to} Nat(B)$ is weaker than a naturalization mapping $nat : D \overset{m}{\to} Nat(B)$ (denote $nat' \leq nat$), if there exists a code computable function $cc : Nat(B) \overset{m}{\to} Nat(B)$ such that $nat' = nat \circ cc$. The introduced relation is reflexive and transitive, but not antisymmetric. Thus, this relation is a preorder.

Theorem 1. *Let $nat, nat' : D \overset{m}{\to} Nat(B)$ be naturalization mappings such that $nat' \leq nat$. Then*

$$NatComp(D, B, nat') \subseteq NatComp(D, B, nat).$$

Proof. Let $f' \in NatComp(D, B, nat')$. Natural computability of f' is based on a certain code computable function g' such that $f' = nat' \circ g' \circ denat'$. From this follows that $f' = (nat \circ cc) \circ g' \circ (nat \circ cc)^{-1} = nat \circ cc \circ g' \circ cc^{-1} \circ nat^{-1} = nat \circ (cc \circ g' \circ cc^{-1}) \circ denat$. If cc^{-1} is code computable, then $cc \circ g' \circ cc^{-1}$ is also code computable. Therefore, $f' \in NatComp(D, B, nat)$. But in general, cc^{-1} is not code computable. This difficulty can be overcome by constructing a code computable function g which simulates $cc \circ g' \circ cc^{-1}$. To construct such a function we have first to consider index computable functions hcc and hg, on which computability of cc and g is based. Then, using these functions hcc and hg, we construct by conventional methods of the recursion theory an index computable function hg, which defines g. From this follows that $f' \in NatComp(D, B, nat)$.

So, if two naturalization mappings nat and nat' are equivalent ($nat \leq nat'$ and $nat' \leq nat$), then these mappings define the same class of naturally computable functions, i.e. $NatComp(D, B, nat') = NatComp(D, B, nat)$.

Having defined the notion of natural computability we can now construct algebraic representations of complete classes of naturally computable partial multi-valued functions for various data structures, which are considered as specializations of nominative data.

5 Complete Classes of Computable Partial Multi-valued Functions

In this short paper we present without details only a few results describing complete classes of computable functions over simple subclasses of nominative data.

We start with the simplest case.

5.1 Computability over Abstract Data

Let D be an abstract set. It means that nothing is known about the structure of its elements. This treatment can be expressed by the naturalization mapping $nat\_a : D \to Nat(D)$ such that $nat\_a(d) = (0, < d >)$ for every $d \in D$.

To define the complete class of naturally computable functions over D, we have to describe all index computable function of the type $Nat^2 \overset{m}{\to} Nat \times Nat^*$. It is easy to understand that under the naturalization mapping nat_a we need to know the results of index computable function only on the element (0,1). There are only three possible behaviours of an index computable function on (0,1):

- to be undefined,
- to yield (0,1),
- or to make non-deterministic choice between being undefined and yielding (0,1).

These three cases induce the following functions of type $D \overset{m}{\to} D$:

- the everywhere undefined function *und*,
- the identity function *id*,
- the non-deterministic function *und-id* such that *und-id*(d) is undefined or is equal to d.

Theorem 2. *The complete class of naturally computable partial multi-valued functions over the abstract set D precisely coincides with the class of functions* {*und, id, und-id*}.

5.2 Computability over Named Data

A class $NAD(V, W)$ of named data is a special subclass of nominative data with single-valued naming and is defined inductively on the basis of the set of names V and the set of basic values W:

- if $w \in W$, then $w \in NAD(V, W)$,
- if $v_1, ..., v_n$ are pairwise distinct names from V, $d_1, ..., d_n \in NAD(V, W)$, then
$$[v_1 \mapsto d_1, ..., v_n \mapsto d_n] \in NAD(V, W).$$

We can also describe the set $NAD(V, W)$ by the recursive definition

$$NAD(V, W) = W \cup (V \overset{n}{\to} NAD(V, W)),$$

where $A \overset{n}{\to} R$ is the set of finite single-valued mappings.

The class of computable functions over $NAD(V, W)$ depends on the abstraction level, on which V and W are considered. Here we present only two cases determined by finite and countable sets of names.

Let $V = \{v_0, ..., v_m\}$ be a finite set ($m > 0$), W be an abstract set. Then data of the set $NAD(V, W)$ are called V-finite W-abstract named data.

This understanding of $NAD(V, W)$ can be presented by the naturalization mapping $nat\_fa : NAD(V, W) \overset{m}{\to} Nat(W)$, which is defined inductively as follows:

– if $d \in W$, then
$$nat\_fa(d) = (c(0, 0), < d >);$$
– if $d = [v_{i_1} \mapsto d_1, ..., v_{i_n} \mapsto d_n],\ i_1 < ... < i_n, n \geq 0,$
$$nat\_fa(d_j) = (k_j, < b_{j_1}, ..., b_{j_{l_j}} >),\ 1 \leq j \leq n,$$
then
$$nat\_fa(d) = (c(1, c(n, c(k'_1, ...c(k'_n, 0)...))), < b_{1_1}, ..., b_{1_l}, ..., b_{n_1}, ..., b_{n_{l_n}} >),$$
where $k'_j = c(i_j, c(k_j, l_j))$, $1 \leq j \leq n$.

Here $c : Nat \times Nat \to Nat$ is the Cantor's pairing function.

Having defined the naturalization mapping for $NAD(V, W)$, we obtain the class $NatComp(D, B, nat\_fa)$ of all naturally computable functions over the class $NAD(V, W)$. The algebraic description of this class is based on a certain algebra, which is called the BACON algebra (BAsic COmposition Nominative algebra [5]). This algebra has the set of functions $NAD(V, W) \overset{m}{\to} NAD(V, W)$ as its carrier set, compositions $\circ$, $*$, ∇ as its binary operations, parametric functions $\Rightarrow v$, $v\Rightarrow$, $v!$ $(v \in V)$ and a function *choice* as its null-ary operations. The class of all terms of this algebra is called the BACON language [5]. It turns out that this language presents the class $NatComp(D, B, nat\_fa)$.

Theorem 3. *The complete class of naturally computable partial multi-valued functions over the set $NAD(V, W)$ of V-finite W-abstract named data precisely coincides with the class of functions obtained by closure of the set of functions $\{\Rightarrow v_0, ..., \Rightarrow v_m, v_0\Rightarrow, ..., v_m\Rightarrow, v_0!, ..., v_m!, choice\}$ under the set of compositions $\{\circ, *, \nabla\}$.*

To prove the theorem, we first have to show, that BACON terms (programs) describe functions, which are naturally computable. This can be easily done in two steps.

1. Prove that functions $\Rightarrow v$, $v\Rightarrow$, $v!$ $(v \in V)$ and *choice* are naturally computable.

2. Prove that compositions $\circ, *, \nabla$ preserve natural computability of functions.

After that, we have to prove that any function $f \in NatComp(D, B, nat\_fa)$ can be presented by some BACON term. This proof consists of the following steps.

1. Represent Nat, Nat^2, Nat^* and $Nat(W)$ within $NAD(V,W)$.

2. Represent by BACON terms all partial recursive functions, index computable functions and code computable functions.

3. Represent by BACON terms the naturalization mapping nat_fa and the denaturalization mapping $denat\_fa$.

The details of these steps can be found in [5].

At last, taking into account that $f = nat\_fa \circ g \circ denat\_fa$ for some code computable function $g : Nat(W) \overset{m}{\to} Nat(W)$ we can conclude that f can be represented by some BACON term. This completes the proof of the theorem.

The obtained result can be generalized for the following class of named data.

Let $V = \{v_0, v_1, ...\}$ be an enumerable set, W be an abstract set ($V \cap W = \emptyset$). Since V is enumerable, any name from V can be recognised and generated. Therefore, elements of the set V will be used not only as names but also as basic values. In other words, we will consider the set of named data $NAD(V, W \cup V)$. Such data are called V-enumerable W-abstract named data.

This understanding of $NAD(V, W \cup V)$ can be presented by the naturalization mapping $nat\_ea : NAD(V, W \cup V) \overset{m}{\to} Nat(W)$, which is defined inductively as follows:

– if $d \in W$, then
$$nat(d) = (c(0,0), < d >);$$
– if $d \in V$, $d = v_i$, then
$$nat(d) = (c(1,i), \emptyset), i = 0, 1, ...;$$
– if $d = [v_{i_1} \mapsto d_1, ..., v_{i_m} \mapsto d_m]$, $i_1 < ... < i_m$, $m \geq 0$, and
$$nat(d_j) = (k_j, < b_{j_1}, ..., b_{j_{l_j}} >),\ j = 1, ..., m,$$
then
$$nat(d) = (c(m+2, c(k'_1, ..., c(k'_m, 0)...)), < b_{1_1}, ..., b_{m_{l_m}} >),$$
where $k'_j = c(i_j, c(k_j, l_j))$, $j = 1, ..., m$.

In order to describe the complete class over $NAD(V, W \cup V)$, we should use the following additional functions.

– Functions over V: successor $succ_V$, predecessor $pred_V$ and constant $\bar{v}_0$.

– Equalities: $=v_0, =\emptyset$.

– Unary predicates: $\in V, \in W$.

– Binary functions: as (naming), cn (denaming), ex (removal) and predicate ec (existence of named component), such that $as(v,d) = \Rightarrow v(d)$, $cn(v,d) = v\Rightarrow(d)$, $ex(v,d) = \backslash v(d)$, $ec(v,d) = v!(d)$ for any $v \in V$, $d \in NAD(V, W \cup V)$.

Theorem 4. *The complete class of naturally computable partial multi-valued functions over the set $NAD(V, W \cup V)$ of V-enumerable W-abstract named data precisely coincides with the class of functions obtained by closure of the set of functions $\{\Rightarrow v_0, \Rightarrow v_1, \in V, \in W, \bar{v}_0, =v_0, =\emptyset, succ_V, pred_V, as, cn, ex, ec, choice\}$ under the set of compositions $\{\circ, *, \nabla\}$.*

5.3 Computability over Nominative Data

In comparison with named data, nominative data allow multi-valued naming. To work efficiently with such data we have to consider a more specific abstraction level introducing equality on W. Nominative data of this level will be called W-equational data. We will consider here only finite nominative data. For such a class of nominative data a naturalization mapping should additionally represent in the code of the resulting natural data the information about all equalities between basic elements of the initial nominative data. The detail are described in [17].

To present computable functions we additionally use a subtraction function $\setminus$ of nominative data and binary summation composition $\sqcup$. In this case the equality on W is derivable. For the class of V-finite W-equational nominative data we can formulate the following result.

Theorem 5. *The complete class of naturally computable partial multi-valued functions over the set of V-finite W-equational nominative data precisely coincides with the class of functions obtained by closure of the set of functions $\{\Rightarrow v_0,\ldots, \Rightarrow v_m,\ v_0\Rightarrow,\ldots,\ v_m\Rightarrow,\ v_0!,\ldots,\ v_m!,\ choice,\ \setminus\}$ under the set of compositions $\{\circ, *, \sqcup\}$.*

5.4 Computability over Sequences

Traditional data structures may be considered as concretizations of nominative data. Here we present the completeness result for functions over sequences.

Let B be an abstract set and $Seq(B)$ be the set of all sequences, hierarchically constructed from elements of B. The set $Seq(B)$ may be defined by recursive definition $Seq(B) = B \cup Seq(B)^*$.

The structure $Seq(B)$ has been investigated in different works. We shall use the notations of [18]. Four new functions are introduced: $first$, $tail$, $apndl$, $is\text{-}atom$. Also, we need a composition, called construction:

$$[f, g](d) =< f(d),\ g(d) > .$$

Theorem 6. *The complete class of naturally computable partial multi-valued functions over the set $Seq(B)$ precisely coincides with the class of functions obtained by closure of the set of functions $\{first, tail, apndl, is\text{-}atom, choice\}$ under the set of compositions $\{\circ, *, [\ ]\}$.*

In a similar way we can also generalize the completeness results for compositional databases presented in [19].

6 Conclusion

In this paper we defined the notion of natural computability for partial multi-valued functions, which represent semantics of non-deterministic programs. This computability is a special abstract computability, that satisfy the main requirements formulated by A.P. Ershov. The complete classes of naturally computable functions are described for simple cases of nominative data. The proposed technique can be used for more rich data structures. The notion of natural computability forms a base for the notion of determinant computability of compositions. The obtained results can be used to study computational completeness of programming, specification and database query languages of various abstraction levels.

References

1. A.P. Ershov. Abstract computability on algebraic structures. In: A.P. Ershov, D. Knuth (Eds) Algorithms in modern mathematics and computer science. Berlin: Springer (1981) 397–420
2. A.P. Ershov. Computability in arbitrary domains and bases. Semiotics and Informatics, No. 19 (1982) 3–58. In Russian.
3. N.S. Nikitchenko. On the construction of classes of generalized computable functions and functionals, UkrNIINTI, techn. report No 856 Uk-84, Kiev (1984) 51 p. In Russian.
4. I.A. Basarab, N.S. Nikitchenko, V.N. Red'ko. Composition databases, Kiev, Lybid' (1992) 192 p. In Russian.
5. N. Nikitchenko. A composition nominative approach to program semantics. Technical Report IT-TR: 1998-020. Technical University of Denmark (1998) 103 p.
6. V.N. Red'ko. Composition of programs and composition programming. Programmirovanie, No 5 (1978) 3–24. In Russian.
7. K. Grue. Map theory. Theoretical Computer Science, v. 102(1) (1992) 1–133
8. The Vienna development method: the meta-language. Ed. by D. Bjorner, C.B. Jones. LNCS, v. 61 (1978) 374 p.
9. M. Atkinson, et. al. The object-oriented database system manifesto. DOOD'89 (1989) 40–57
10. R. Gandy. Church's thesis and principles for mechanisms. The Kleene Symp. Eds. J. Barwise, et. al, Amsterdam: North-Holland (1980) 123–148
11. D. Scott. Domains for denotational semantics. LNCS, v. 140 (1982) 577–613
12. Y.N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. Lect. Notes Math, v. 1104 (1984) 289-362
13. A.J. Kfoury, P. Urzyczyn. Necessary and sufficient conditions for the universality of programming formalism. Acta Informatica, v. 22 (1985) 347–377
14. E. Dahlhaus, J. Makowsky. The Choice of programming primitives for SETL-like programming languages. LNCS, v. 210 (1986) 160–172
15. J.V. Tucker, J.I. Zucker. Deterministic and nondeterministic computation, and Horn programs, on abstract data types. J. Logic Programming, v. 13 (1992) 23–55
16. V.Yu. Sazonov. Hereditarily-finite sets, data bases and polynomial computability. Theoretical Computer Science, v. 119 (1993) 187–214
17. N.S. Nikitchenko. Construction of composition systems on a base of identified data. Kibernetika i systemny analiz, No 6 (1995) 38–44. In Russian.
18. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communs. ACM, v. 21 (1978) 613-641
19. I.A. Basarab, B.V. Gubsky, N.S. Nikitchenko, V.N. Red'ko. Composition models of databases. Extending Inf. Syst. Technology, II Int. East-West Database Workshop, Sept. 25-28, 1994, Klagenfurt, Austria (1994) 155-163

On Lexicographic Termination Ordering with Space Bound Certifications

Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen

Loria, Calligramme project
B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
{Guillaume.Bonfante,Jean-Yves.Marion,Jean-Yves.Moyen}@loria.fr

Abstract. We propose a method to analyse the program space complexity, based on termination orderings. This method can be implemented to certify the runspace of programs. We demonstrate that the class of functions computed by first order functional programs over free algebras which terminate by Lexicographic Path Ordering and admit a polynomial quasi-interpretation, is exactly the class of functions computable in polynomial space.

1 Introduction

Motivations. There are several motivations to develop automatic program complexity analysis:

1. The control of the resources consumed by programs is a necessity, in software development.
2. There is a growing interest in program resource certifications. For example, Benzinger [2] has implemented a prototype to certify the time complexity of programs extracted from Nuprl [6]. Various systems have been defined to control resources in functional languages, see Weirich and Crary [7] and Hofmann [12].
3. Our approach is based on well-known termination orderings, used in term rewriting systems, which are easily implemented.
4. The study of program complexity is of a different nature compared to program verification or termination. It is not enough to know what is computed, we have to know how it is performed. So, it gives rise to interesting questions whose answers might belong to a theory of feasible algorithms which is not yet well established.

Our Results. We consider first order functional programs over any kind of constructors, which terminate by Lexical Path Ordering (LPO). We demonstrate that the class of functions which are computed by LPO-programs admitting polynomially bounded quasi-interpretations, is exactly the class of functions which are computed in polynomial space. (See Section 3.2 for the exact statement.) This resource analysis can be partialy automatized. Indeed, Krishnamoorthy and Narendran in [16] have proved that termination by LPO is NP-complete. To find a quasi-interpretation is not too difficult in general, because the program denotation turns out to be a good candidate.

Complexity and Termination Orderings. Termination orderings give rise to interesting theoretical questions concerning the classes of functions for which they provide termination proofs. Weiermann [23] has shown that LPO characterizes the multiple recursive functions and Hofbauer [10] has shown that Multiset Path Ordering (MPO) gives rise to a characterization of primitive recursive functions. While both of these contain functions which are highly unfeasible, the fact remains that many feasible algorithms can be successfully treated using one or both. Quasi-interpretations allows us to tame the complexity of treated algorithms. Indeed, it has been established [21] that functions computed by programs terminating by MPO and admitting a polynomial quasi-interpretation are exactly the polynomial time computable functions. This last result might be compared with the one of Hofbauer. Analogously, the result presented in this paper might be compared with Weiermann's one.

Other Related Characterizations of Poly-Space. There are several characterizations of PSPACE in Finite Model Theory, and we refer to Immerman's book [13] for a complete presentation. A priori, Finite Model Theory approach is not relevant from the point of view of programming languages because computational domains are infinite. But recently, Jones [14] has showed that polynomial space languages are characterized by mean of read-only functional programs. He has observed a closed relationship with a characterization of Goerdt [9].

On infinite computational domains, characterizations of complexity classes go back to Cobham's seminal work [5]. The set of polynomial space computable functions has been identified by Thompson [22]. Those characterizations are based on bounded recursions. Hence, they do not directly study algorithms and so they are not relevant to automatic complexity analysis.

Lastly, from the works of Bellantoni and Cook [1] and of Leivant [18], purely syntactic characterizations of polynomial space computable functions were obtained in [19,20]. The underlying principle is the concept of ramified recursion. From the point of view of automatic complexity analysis, the main interest of this approach is that we have syntactic criteria to determine the complexity of a program. However, a lot of algorithms are ruled out. Several solutions have been proposed to enlarge the class of algorithms captured. Hofmann [11] has proposed a type system with modalities to deal with non-size increasing functions, e.g. the functions *max* and *min*. Another solution is to introduce a ramified version of termination orderings MPO [20], which delineates polynomial time and polynomial space computable functions.

2 First Order Functional Programming

Throughout the following discussion, we consider three disjoint sets $\mathcal{X}, \mathcal{F}, \mathcal{C}$ of variables, function symbols and constructors.

2.1 Syntax of Programs

Definition 1. *The sets of terms, patterns and function rules are defined in the following way:*

(Constructor terms)	$\mathcal{T}(\mathcal{C}) \ni u$	$::= \mathbf{c} \mid \mathbf{c}(u_1, \dots, u_n)$
(Ground terms)	$\mathcal{T}(\mathcal{C}, \mathcal{F}) \ni s$	$::= \mathbf{c} \mid \mathbf{c}(s_1, \dots, s_n) \mid f(s_1, \dots, s_n)$
(terms)	$\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t$	$::= \mathbf{c} \mid x \mid \mathbf{c}(t_1, \dots, t_n) \mid f(t_1, \dots, t_n)$
(patterns)	$\mathcal{P} \ni p$	$::= \mathbf{c} \mid x \mid \mathbf{c}(p_1, \dots, p_n)$
(rules)	$\mathcal{D} \ni d$	$::= f(p_1, \cdots, p_n) \to t$

where $x \in \mathcal{X}$, $f \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$[1]*. The size $|t|$ of a term t is the number of symbols in t.*

Definition 2. *A program $\mathcal{E}(\mathtt{main})$ is a set $\mathcal{E}$ of $\mathcal{D}$-rules such that for each rule $f(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$, each variable in t appears also in some pattern p_i. All along the paper, we assume that the set of rules is implicit and we just write $\mathtt{main}$ to denote the program.*

Example 1. The following program is intended to compute the Ackermann's function. Take the set of constructors to be $\mathcal{C} = \{\mathbf{0}, \mathbf{S}\}$. We won't explicitly note the arity of symbols, as it is implicitly given by the rules.

$$\begin{aligned} \mathtt{Ack}(\mathbf{0}, n) &\to \mathbf{S}(n) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{0}) &\to \mathtt{Ack}(m, \mathbf{S}(\mathbf{0})) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{S}(n)) &\to \mathtt{Ack}(m, \mathtt{Ack}(\mathbf{S}(m), n)) \end{aligned}$$

2.2 Semantics

The signature $\mathcal{C} \cup \mathcal{F}$ and the set $\mathcal{E}$ of rules induce a rewrite system which brings us the operational semantics. We recall briefly some vocabulary of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [8] from which we take the notations. The rewriting relation $\to$ induced by a program $\mathtt{main}$ is defined as follows $t \to s$ if s is obtained from t by applying one of the rules of $\mathcal{E}$. The relation $\overset{*}{\to}$ is the reflexive-transitive closure of $\to$. Lastly, $t \overset{!}{\to} s$ means that $t \overset{*}{\to} s$ and s is in normal form, *i.e.* no other rule may be applied. A ground (resp. constructor) substitution is a substitution from $\mathcal{X}$ to $\mathcal{T}(\mathcal{C}, \mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$).

We now give the semantics of confluent programs, that is programs for which the associated rewrite system is confluent. The domain of interpretation is the constructor algebra $\mathcal{T}(\mathcal{C})$.

Definition 3. *Let $\mathtt{main}$ be a confluent program. The function computed by $\mathtt{main}$ is a partial function $[\![\mathtt{main}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ where n is the arity of $\mathtt{main}$. For all $u_i \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{main}]\!](u_1, \dots, u_n) = v$ iff $\mathtt{main}(u_1, \dots, u_n) \overset{!}{\to} v$ with $v \in \mathcal{T}(\mathcal{C})$. Note that due to the form of the rules a constructor term is a normal form ; as the program is confluent, it is uniquely defined. Otherwise, that is if there is no such normal form, $[\![\mathtt{main}]\!](u_1, \dots, u_n)$ is undefined.*

[1] We shall use type writer font for function symbol and bold face font for constructors.

3 LPO and Quasi-Interpretations

3.1 Lexicographic Path Ordering

Termination orderings are widely used to prove the termination of term rewriting systems. The *Lexicographic Path Ordering* (LPO) is one of them, it was introduced by Kamin and Lévy [15]. We briefly describe it, together with some basic properties we shall use later on.

Definition 4. *Let $\prec$ be a term ordering. We note $\prec^l$ its lexicographic extension. A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is a quasi-ordering (ordering) on the set $\mathcal{F}$ of function symbols. It is canonically extended on $\mathcal{C} \cup \mathcal{F}$ by saying that constructors are smaller than functions. Given such a precedence, the lexicographic path ordering $\prec_{lpo}$ is defined recursively by the rules:*

$$\frac{s \preceq_{lpo} t_i}{s \prec_{lpo} f(\dots, t_i, \dots)} \; f \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{s_i \prec_{lpo} f(t_1, \dots, t_n) \qquad g \prec_{\mathcal{F}} f}{g(s_1, \dots, s_m) \prec_{lpo} f(t_1, \dots, t_n)} \; g \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1, \dots, s_n) \prec^l_{lpo} (t_1, \dots, t_n) \qquad f \approx_{\mathcal{F}} g \qquad s_j \prec_{lpo} f(t_1, \dots, t_n)}{g(s_1, \dots, s_n) \prec_{lpo} f(t_1, \dots, t_n)}$$

Definition 5. *$\lhd$ is the usual subterm ordering. That is $s \lhd f(t_1, \dots, t_n)$ if and only if $s = t_i$ or $s \lhd t_i$ for some $1 \leq i \leq n$.*

Lemma 1. *Let t and s be constructor terms. $s \prec_{lpo} t$ implies $|s| < |t|$.*

Example 2. One can verify that the Ackermann's function of example 1 terminates by LPO.

3.2 Polynomial Quasi-Interpretation

Definition 6. *Let $f \in \mathcal{F} \bigcup \mathcal{C}$ be either a function symbol or a constructor of arity n. A quasi-interpretation of f is a mapping $(\!| f |\!) : \mathbf{N}^n \to \mathbf{N}$ which satisfies (i) $(\!| f |\!)$ is (not necessarily strictly) increasing with respect to each argument, (ii) $(\!| f |\!)(X_1, \dots, X_n) \geq X_i$, for all $1 \leq i \leq n$, (iii) $(\!| f |\!) > 0$ for each 0-ary symbol $f \in \mathcal{F} \cup \mathcal{C}$.*

We extend a quasi-interpretation $(\!| - |\!)$ to terms canonically: $(\!| f(t_1, \dots, t_n) |\!) = (\!| f |\!)((\!| t_1 |\!), \cdots, (\!| t_n |\!))$

Definition 7. *$(\!| - |\!)$ is a quasi-interpretation of a program `main` if for each rule $l \to r \in \mathcal{E}($`main`$)$ and for each closed substitution σ, $(\!| r\sigma |\!) \leq (\!| l\sigma |\!)$.*

Lemma 2. *If t and t' are two terms such that $t \to t'$, then $(\!| t |\!) \geq (\!| t' |\!)$.*

Definition 8. *A program `main` admits a polynomial quasi-interpretation $(\!| - |\!)$, if $(\!| - |\!)$ is bounded by a polynomial.*

A polynomial quasi-interpretation is said to be of kind 0 *if for each constructor $\mathbf{c}$, $(\!| \mathbf{c} |\!)(X_1, \dots, X_n) = a + \sum_{i=1}^{n} X_i$ for some constant $a > 0$.*

Remark 1. Quasi-interpretations are not sufficient to prove program termination. Indeed, the rule $\mathtt{f}(x) \to \mathtt{f}(x)$ admits a quasi-interpretation but doesn't terminate.

Unlike quasi-interpretation, a program interpretation satisfies to extra conditions: (i) $(\!|f|\!)(X_1, \dots, X_n) > X_i$, (ii) $(\!|r\sigma|\!) < (\!|l\sigma|\!)$. Programs admitting an interpretation terminate. This sort of termination proof, by polynomial interpretations, was introduced by Lankford [17]. Bonfante, Cichon, Marion and Touzet [3] proved that programs admitting interpretation of kind 0 are computable in polynomial time.

Definition 9. *A LPO-program is a program that terminates by LPO.*

A LPO$^{Poly(0)}$-program is a LPO-program that admits a quasi-interpretation of kind 0.

Theorem 1 (Main Result). *The set of functions computed by LPO$^{Poly(0)}$-programs is exactly the set of funtions computable in polynomial space.*

Proof. It is a consequence of Theorem 2 and Theorem 4.

Example 3.

1. The Ackermann's function of example 1 doesn't admit a polynomial quasi-interpretation because $[\![\mathtt{Ack}]\!]$ is not polynomially bounded.
2. The Quantified Boolean Formula (QBF) is the problem of the validity of a boolean formula with quantifiers over propositional variables. It is well-known to be PSPACE complete. Wlog, we restrict formulae to $\neg, \vee, \exists$. It can be solved by the following rules:

$$\begin{array}{rlcrl}
\mathtt{not}(\mathbf{tt}) & \to \mathbf{ff} & \quad & \mathbf{0} = \mathbf{0} & \to \mathbf{tt} \\
\mathtt{not}(\mathbf{ff}) & \to \mathbf{tt} & & \mathbf{S}(x) = \mathbf{0} & \to \mathbf{ff} \\
\mathtt{or}(\mathbf{tt}, x) & \to \mathbf{tt} & & \mathbf{0} = \mathbf{S}(y) & \to \mathbf{ff} \\
\mathtt{or}(\mathbf{ff}, x) & \to x & & \mathbf{S}(x) = \mathbf{S}(y) & \to x = y
\end{array}$$

$$\begin{aligned}
\mathtt{main}(\phi) &\to \mathtt{ver}(\phi, \mathbf{nil}) \\
\mathtt{in}(x, \mathbf{nil}) &\to \mathbf{ff} \\
\mathtt{in}(x, \mathbf{cons}(a, l)) &\to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{aligned}$$

In the next function, t is the set of variables whose value is $\mathbf{tt}$.

$$\begin{aligned}
\mathtt{ver}(\mathbf{Var}(x), t) &\to \mathtt{in}(x, t) \\
\mathtt{ver}(\mathbf{Or}(\phi_1, \phi_2), t) &\to \mathtt{or}(\mathtt{ver}(\phi_1, t), \mathtt{ver}(\phi_2, t)) \\
\mathtt{ver}(\mathbf{Not}(\phi), t) &\to \mathtt{not}(\mathtt{ver}(\phi, t)) \\
\mathtt{ver}(\mathbf{Exists}(n, \phi), t) &\to \mathtt{or}(\mathtt{ver}(\phi, \mathbf{cons}(n, t)), \mathtt{ver}(\phi, t))
\end{aligned}$$

These rules are ordered by LPO by putting $\{\mathtt{not}, \mathtt{or}, \_\mathtt{=}\_\} \prec_{\mathcal{F}} \mathtt{in} \prec_{\mathcal{F}} \mathtt{ver} \prec_{\mathcal{F}} \mathtt{main}$.

They admit the following quasi-interpretations:
- $(\!|\mathtt{c}|\!)(X_1, \dots, X_n) = 1 + \sum_{i=1}^{n} X_i$, for each n-ary constructor,
- $(\!|\mathtt{ver}|\!)(\Phi, T) = \Phi + T$, $(\!|\mathtt{main}|\!)(\Phi) = \Phi + 1$,
- $(\!|\mathtt{f}|\!)(X_1, \dots, X_n) = \max_{i=1}^{n} X_i$, for the other function symbols.

4 LPO$^{\boldsymbol{Poly(0)}}$-Programs Are PSPACE Computable

Definition 10. *A* state *is a tuple* $\langle \mathtt{f}, t_1, \dots, t_n\rangle$ *where* $\mathtt{f}$ *is a function symbol of arity* n *and* $t_1, \dots, t_n$ *are constructor terms.*

State$(\mathtt{main})$ *is the set of all states built from the symbols of a program* $\mathtt{main}$. *State*$^A(\mathtt{main}) = \{\langle \mathtt{f}, t_1, \dots, t_n\rangle \in State(\mathtt{main})\ /\ |t_i| < A\}$. *Intutively, a state represents a recursive call in the evaluation process.*

Definition 11. *Let* $\mathtt{main}$ *be a LPO*$^{Poly(0)}$*-program,* $\eta_1 = \langle \mathtt{f}, t_1, \dots, t_n\rangle$ *and* $\eta_2 = \langle \mathtt{g}, s_1, \dots, s_m\rangle$ *be two states of State*$(\mathtt{main})$. *A* transition *is a triplet* $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ *such that:*

(i) $e \equiv \mathtt{f}(p_1, \dots, p_n) \to t$
(ii) there is a substitution σ *such that* $p_i\sigma = t_i$ *for all* $1 \le i \le n$
(iii) there is a subterm $\mathtt{g}(u_1, \dots, u_m) \trianglelefteq t$ *such that* $u_i\sigma \overset{!}{\to} s_i$ *for all* $1 \le i \le n$.

Transition$(\mathtt{main})$ *is the set of all transitions between the elements of State*$(\mathtt{main})$.

$\overset{*}{\rightsquigarrow}$ *is the reflexive transitive closure of* $\cup_{e\in\mathcal{E}} \overset{e}{\rightsquigarrow}$.

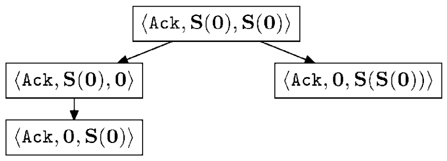
Definition 12. *Let* $\mathcal{E}(\mathtt{main})$ *be a LPO*$^{Poly(0)}$*-program and* $\langle \mathtt{f}, t_1, \dots, t_n\rangle$ *be a state (*i.e. $\langle \mathtt{f}, t_1, \dots, t_n\rangle \in State(\mathtt{main})$*). A* $\langle \mathtt{f}, t_1, \dots, t_n\rangle$*-call tree* τ *is defined as follows:*

– *The root of* τ *is* $\langle \mathtt{f}, t_1, \dots, t_n\rangle$.
– *For each node* η_1*, the children of* η_1 *is exactly the set of states* $\{\eta_2 \in State(\mathtt{main})\ /\ \eta_1 \overset{e}{\rightsquigarrow} \eta_2 \in Transition(\mathtt{main})\}$ *where* e *is a given equation of* $\mathcal{E}$.

CT$(\langle \mathtt{f}, t_1, \dots, t_n\rangle)$ *is the set of all* $\langle \mathtt{f}, t_1, \dots, t_n\rangle$*-call trees.*

$$CT^A(\langle \mathtt{f}, t_1, \dots, t_n\rangle) = \{\tau \in CT(\langle \mathtt{f}, t_1, \dots, t_n\rangle)\ /\ \forall \eta \in \tau, \eta \in State^A(\mathtt{main})\}$$

Example 4. The (unique) $\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0})\rangle$-call tree of $CT(\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0})\rangle)$ is:


$$\begin{tikzcd}
& \boxed{\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0})\rangle} \arrow[dl] \arrow[dr] & \\
\boxed{\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{0}\rangle} \arrow[d] & & \boxed{\langle \mathtt{Ack}, \mathbf{0}, \mathbf{S}(\mathbf{S}(\mathbf{0}))\rangle} \\
\boxed{\langle \mathtt{Ack}, \mathbf{0}, \mathbf{S}(\mathbf{0})\rangle} & &
\end{tikzcd}$$

Lemma 3. *Let* $\mathtt{main}$ *be a LPO-program,* α *be the number of function symbols in* $\mathtt{main}$ *and* d *be the maximal arity of a function symbol. The following facts hold for all* $\tau \in CT^A(\langle \mathtt{f}, t_1, \dots, t_n\rangle)$*:*

1. *If* $\langle \mathtt{f}, t_1, \dots, t_n\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, s_1, \dots, s_m\rangle$ *then (a)* $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$ *or (b)* $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ *and* $(s_1, \dots, s_m) \prec^l_{lpo} (t_1, \dots, t_n)$.

2. *If $\langle f, t_1, \ldots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \ldots, s_m\rangle$ in τ and $\mathtt{g} \approx_{\mathcal{F}} f$ then the number of states between $\langle f, t_1, \ldots, t_n\rangle$ and $\langle g, s_1, \ldots, s_m\rangle$ is bounded by A^d.*
3. *The length of each branch of τ is bounded by $\alpha \times A^d$.*

Proof.

1. Because the rules of the program decrease for LPO.
2. Let $\langle \mathtt{h}, u_1, \ldots, u_p\rangle$ be a child of $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$. As t_i and u_j are constructor terms, due to first point of the current lemma and lemma 1, we have $(|u_1|, \cdots, |u_p|) <^l (|t_1|, \cdots, |t_n|)$. Since the size of each component is bounded by A, the length of the decreasing chain is bounded by A^d.
3. In each branch, there are at most A^d states whose function symbols have the same precedence, then A^d states whose function symbol have the precedence immediatly below, and so on. As there are only α function symbols the length of the branch is bounded by $\alpha \times A^d$.

Lemma 4. *Let* **main** *be a $LPO^{Poly(0)}$-program, f be a function symbol and $t_1, \ldots, t_n$ be constructor terms. $CT(f, t_1, \ldots, t_n) = CT^{(\!|f(t_1,\cdots,t_n)|\!)}(f, t_1, \ldots, t_n)$.*

Proof. Let $\tau \in CT(\mathtt{f}, t_1, \ldots, t_n)$ and $\langle \mathtt{g}, s_1, \ldots, s_m\rangle$ be a state in τ. As s_i is a constructor term, $|s_i| \leq (\!|s_i|\!) \leq (\!|\mathtt{g}(s_1, \ldots, s_m)|\!) \leq (\!|\mathtt{f}(t_1, \ldots, t_n)|\!)$ because of the definition of quasi-interpretations.

Lemma 5. *Given a term $t \in \mathcal{T}(\mathcal{C})$, the following holds: $(\!|t|\!) \leq c.|t|$ for some constant c. As a corollary, we have $(\!|main(t_1, \ldots, t_n)|\!) \leq P(\max_{i=1}^n |t_i|)$ for some polynomial P.*

Theorem 2. *Let* **main** *be a $LPO^{Poly(0)}$-program. For each constructor terms $t_1, \ldots, t_n$, the space used by a call by value interpreter to compute* **main**$(t_1, \ldots, t_n)$ *is bounded by a polynomial in $max_i\{|t_i|\}$. Such an interpreter is described in annex.*

Proof. Put $A = (\!|\mathtt{main}(t_1, \ldots, t_n)|\!)$.

The interpreter only needs to store the call stack of each recursive call and the intermediate terms (e_i, b) of the computation. The size of e_i and b are both bounded by A. Note that the computation can be followed on a $\langle \mathtt{main}, t_1, \ldots, t_n\rangle$ call-tree. Each recursive call corresponds a transition on the call-tree. So, the maximal depth of the stack corresponds to the maximal length of the branch in a call tree of $CT\langle \mathtt{main}, t_1, \ldots, t_n\rangle = CT^A\langle \mathtt{main}, t_1, \ldots, t_n\rangle$. So it is bounded by $\alpha \times A^d$ (see Lemma 3(3)). The values stored in the stack are states; as a consequence, the size of each of them is bounded by $d \times A + O(1)$.

Therefore, the space used by the interpreter is bounded by $\alpha \times d \times A^{d+1} + A + O(1)$, and A is a polynomial in the size of $\max_i\{|t_i|\}$ by Lemma 5.

5 Parallel Register Machines over Words

We present here a restriction of Parallel Register Machines introduced in [19]. They are an adaptation of the alternating Turing machine. Chandra, Kozen and Stockmeyer have demonstrated that the set of functions that alternating Turing machines computes in polynomial time is exactly the state of polynomial space computable functions.

Definition 13. $\mathbb{W} = \mathcal{T}(\{\boldsymbol{0}^1, \boldsymbol{1}^1, \boldsymbol{\epsilon}^0\})$. $\mathbb{N} = \mathcal{T}(\{\boldsymbol{s}^1, \diamond^0\})$.

Note that $\mathbb{W}$ *(resp.* $\mathbb{N}$*) is isomorphic to* $\{\boldsymbol{0}, \boldsymbol{1}\}^*$ *(resp. natural numbers), both sets are used indiferently in the rest of the section.*

Definition 14. *A Parallel Register Machine (PRM) over the word algebra* $\mathbb{W}$ *consists in:*

1. *a finite set* $S = \{s_0, s_1, \dots, s_k\}$ *of* states*, including a distinct state* BEGIN.
2. *a finite list* $\Pi = \{\pi_1, \dots, \pi_m\}$ *of* registers*; we write* OUTPUT *for* π_m*; Registers will only store values in* $\mathbb{W}$*;*
3. *an ordering* $\blacktriangleleft$ *on* $\mathbb{W}$*:* $\boldsymbol{\epsilon} \blacktriangleleft y$, $\boldsymbol{0}(x) \blacktriangleleft \boldsymbol{1}(y)$, $\boldsymbol{i}(x) \blacktriangleleft \boldsymbol{i}(y)$ *if and only if* $x \blacktriangleleft y$.
4. *a function com mapping states to* commands *which are:*
   $[\boldsymbol{Succ}(\pi' = \boldsymbol{i}(\pi), s')]$, $[\boldsymbol{Pred}(\pi' = \mathbf{p}(\pi), s')]$, $[\boldsymbol{Branch}(\pi, s', s'')]$, $[\boldsymbol{Fork}_{min}(s', s'')]$, $[\boldsymbol{Fork}_{max}(s', s'')]$, $[\boldsymbol{End}]$.

A *configuration* of a PRM M is given by a pair (s, F) where $s \in S$ and F is a function $\Pi \to \mathbb{W}$. We note $[u_1, \dots, u_m]$ for the function which maps $\pi_i \mapsto u_i$ and $\{\pi_i \leftarrow a\}[u_1, \dots, u_m]$ denotes $[u_1, \dots, u_{i-1}, a, u_{i+1}, \dots, u_m]$.

Definition 15. *Given* M *as above we define a semantic partial-function* $\texttt{eval} : \mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$*, that maps the result of the machine in a "time bound" given by the first argument.*

- $\texttt{eval}(0, s, F)$ *is undefined.*
- *If* $com(s)$ *is* $\boldsymbol{Succ}(\pi' = \boldsymbol{i}(\pi), s')$ *then* $\texttt{eval}(t+1, s, F) = \texttt{eval}(t, s', \{\pi' \leftarrow \boldsymbol{i}(\pi)\}F)$*. Note that on the right of the left arrow,* π *denotes the content of the register;*
- *If* $com(s)$ *is* $\boldsymbol{Pred}(\pi' = \mathbf{p}(\pi), s')]$*, then* $\texttt{eval}(t+1, s, F) = \texttt{eval}(t, s', \{\pi' \leftarrow \mathbf{p}(\pi)\}F)$*;*
- *If* $com(s)$ *is* $\boldsymbol{Branch}(\pi, s', s'')$ *then* $\texttt{eval}(t+1, s, F) = \texttt{eval}(t, r, F)$*, where* $r = s'$ *if* $\pi = \boldsymbol{0}(w)$ *and* $r = s''$ *if* $\pi = \boldsymbol{1}(w)$*;*
- *If* $com(s)$ *is* $\boldsymbol{Fork}_{\min}(s', s'')$ *then* $\texttt{eval}(t+1, s, F) = \min_{\blacktriangleleft}(\texttt{eval}(t, s', F), \texttt{eval}(t, s'', F))$*;*
- *If* $com(s)$ *is* $\boldsymbol{Fork}_{max}(s', s'')$ *then* $\texttt{eval}(t+1, s, F) = \max_{\blacktriangleleft}(\texttt{eval}(t, s', F), \texttt{eval}(t, s'', F))$*;*
- *If* $com(s)$ *is* $\boldsymbol{End}$ *then* $\texttt{eval}(t+1, s, F) = F(\text{OUTPUT})$.

Definition 16. *Given a function* $T : \mathbb{N} \to \mathbb{N}$*, we say that the PRM* M *computes* $f : \mathbb{W}^k \to \mathbb{W}$ *in time* T *(or equivalently that* f *is* T*-computable) if for all* $(w_1, \dots, w_k) \in \mathbb{W}^k$*, we have*

$$\texttt{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, [w_1, \dots, w_k, \boldsymbol{\epsilon}, \dots, \boldsymbol{\epsilon}]) = f(w_1, \dots, w_k)$$

Theorem 3 (Chandra & *al* [4]). *Let* $f : \mathbb{W} \to \mathbb{W}$*.* f *is computable in polynomial space iff* f *is PRM-computable in polynomial time.*

5.1 Simulation of PRMs by LPO$^{Poly(0)}$-Programs

The simulation of PRM is done simply by following the rules of the operational semantics of PRM we gave above. In particular, the first argument of `eval` represents a clock.

Lemma 6 (Plug and Play Lemma). *Let $f : \mathbb{W} \to \mathbb{W}$ be a T-time PRM-computable function, then, the function f' is computable by an LPO$^{Poly(0)}$.*

$$\begin{array}{ll} f' : \mathbb{N} \times \mathbb{W} \to \mathbb{W} & \\ \quad (n, w) \mapsto f(w) & \textit{if } n > T(|w|) \\ \quad (n, w) \mapsto \bot & \textit{otherwise} \end{array}$$

Proof. Let the set of constructors be $\mathcal{C} = \{\mathbf{0}, \mathbf{1}, \mathbf{s}, \diamond, \epsilon\} \cup \mathcal{S}$ where S is the set of states. We let the rules in appendix B. Simply, let's say that the functions symbols are: `min`, `max` corresponding to $\min_{\blacktriangleleft}, \max_{\blacktriangleleft}$, `eval` which simulate the rules of the operational semantics and `f'`. We develop here two rules for `eval`.

- $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{Eval}(t, s', \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \ldots, \pi_m)$ if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,
- $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{min}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, ..., \pi_m))$ if $com(s) = \mathbf{Fork}_{\min}(s', s'')$

Take the precedence $\{\texttt{min}, \texttt{max}\} \prec_{\mathcal{F}} \texttt{Eval}$, these rules decrease according to LPO because the time bound decrease. They admit the following quasi-interpretation:

$$\begin{array}{lll} (\!|\boldsymbol{\epsilon}|\!) = 1 & (\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\texttt{min}|\!)(W, W') = \max(W, W') \\ (\!|\diamond|\!) = 1 & (\!|\mathbf{1}|\!)(X) = X + 1 & (\!|\texttt{max}|\!)(W, W') = \max(W, W') \\ & (\!|\mathbf{s}|\!)(X) = X + 1 & \end{array}$$

$$\forall s \in S, (\!|s|\!) = 1 \qquad (\!|\texttt{Eval}|\!)(T, S, \Pi_1, \ldots, \Pi_m) = \max\{\Pi_1, \ldots, \Pi_m\} \times T + S$$

Now, the function `f'` is defined by $\texttt{f'}(n, w) \to \texttt{Eval}(n, \text{BEGIN}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon})$. It is routine to check that $f' = [\![\texttt{f'}]\!]$. This rule decreases by LPO by taking $\texttt{f'} \succ \texttt{Eval}$. There is also a quasi-interpretation for the rule: $(\!|\texttt{f'}|\!)(N, X) = N \times X + 1$.

Lemma 7. *Given $w \in \mathbb{W}$ and a polynomial P, the function $w \in \mathbb{W} \mapsto P(|w|)$ is computable by a LPO$^{Poly(0)}$-program. The proof is given in the appendix.*

Theorem 4. *A polynomial time PRM computable function f can be computed by an LPO$^{Poly(0)}$-program.*

Proof. Follows from Lemma 6 and Lemma 7.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
2. R. Benzinger. *Automated complexity analysis of NUPRL extracts.* PhD thesis, Cornell University, 1999.
3. G. Bonfante, A. Cichon, J-Y Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384, 1999.
4. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
5. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.
6. R. Constable and al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.
7. K. Crary and S. Weirich. Ressource bound certification. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 184 – 198, 2000.
8. N. Dershowitz and J-P Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (North-Holland), 1990.
9. A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1):45–66, 1992.
10. D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.
11. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.
12. M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.
13. N. Immerman. *Descriptive Complexity.* Springer, 1999.
14. N. Jones. The Expressive Power of Higher order Types or, Life without CONS. to appear, 2000.
15. S. Kamin and J-J Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note.
16. M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40(2-3):323–328, October 1985.
17. D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report MTP-3, Louisiana Technical University, 1979.
18. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
19. D. Leivant and J-Y Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz,Poland, 1995. Springer.

20. J-Y Marion. Complexité implicite des calculs, de la théorie à la pratique, 2000. Habilitation.
21. J-Y Marion and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.
22. D.B. Thompson. Subrecursiveness : machine independent notions of computability in restricted time and storage. *Math. System Theory*, 6:3–15, 1972.
23. A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.

A Interpreter for LPO$^{Poly(0)}$-Programs

A space-economical interpreter:

```
function eval($\mathtt{f}, t_1, \dots, t_n$)
begin
    let $e = \mathtt{f}(p_1, \dots, p_n) \to t$
    let $\sigma$ such that $p_i\sigma = t_i$
    let $e_0 = t\sigma$
    let $i = 0$
    foreach $\mathtt{g}(s_1, \dots, s_m) \trianglelefteq e_i \wedge s_j \in \mathcal{T}(\mathcal{C}), j \in \{1, .., m\}$ do
        $b = \text{eval}(\mathtt{g}, s_1, \dots, s_m)$
        $e_{i+1} = e_i\{\mathtt{g}(s_1, \dots, s_m) \leftarrow b\}$
        $i = i + 1$
    return $e_i$
end.
```

where $e_i\{\mathtt{g}(s_1, \dots, s_m) \leftarrow b\}$ denotes the term e_i where each occurence of $\mathtt{g}(s_1, \dots, s_m)$ has been replaced by b.

B Simulation of PRM by LPO$^{Poly(0)}$-Programs

One follows the operational semantics of PRMs.

$$\begin{array}{rlrl}
\mathtt{min}(\epsilon, w) &\to \epsilon & \mathtt{max}(\epsilon, w) &\to w \\
\mathtt{min}(w, \epsilon) &\to \epsilon & \mathtt{max}(w, \epsilon) &\to w \\
\mathtt{min}(\mathbf{0}(w), \mathbf{1}(w')) &\to \mathbf{0}(w) & \mathtt{max}(\mathbf{0}(w), \mathbf{1}(w')) &\to \mathbf{1}(w') \\
\mathtt{min}(\mathbf{1}(w), \mathbf{0}(w')) &\to \mathbf{0}(w') & \mathtt{max}(\mathbf{1}(w), \mathbf{0}(w')) &\to \mathbf{1}(w) \\
\mathtt{min}(\mathbf{i}(w), \mathbf{i}(w')) &\to \mathbf{i}(\mathtt{min}(w, w')) & \mathtt{max}(\mathbf{i}(w), \mathbf{i}(w')) &\to \mathbf{i}(\mathtt{max}(w, w'))
\end{array}$$

with $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$. We have $[\![\mathtt{min}]\!] = \min_{\blacktriangleleft}$ and $[\![\mathtt{max}]\!] = \max_{\blacktriangleleft}$.

(a) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \dots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \dots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,

(b) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \mathbf{i}(\pi'_j), \dots, \pi_m)$
$\to \mathtt{Eval}(t, s', \pi_1, \dots, \pi'_j, \dots, \mathbf{i}(\pi'_j), \dots, \pi_m)$
if $com(s) = \mathbf{Pred}(\pi_k = \mathbf{p}(\pi_j), s')$,

(c) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_j), \pi_{j+1}, \ldots, \pi_m) \rightarrow \mathtt{Eval}(t, r, \pi_1, \ldots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$ where $r = s'$ if $\mathbf{i} = \mathbf{0}$ and $r = s''$ if $\mathbf{i} = \mathbf{1}$,

(d) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m)$
$\rightarrow \mathtt{min}(\mathtt{Eval}(t, s', \pi_1, \ldots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\min}(s', s'')$

(e) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m)$
$\rightarrow \mathtt{max}(\mathtt{Eval}(t, s', \pi_1, \ldots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\max}(s', s'')$

(f) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \rightarrow \pi_m$
if $com(s) = \mathbf{End}$

B.1 Computation of Polynomials by LPO$^{Poly(0)}$-Programs

Each polynomials can be computed with a combination of `add` and `mult`.

$$\begin{aligned} \mathtt{add}(\diamond, y) &\rightarrow y & \mathtt{mult}(\diamond, y) &\rightarrow \diamond \\ \mathtt{add}(\mathtt{s}(x), y) &\rightarrow \mathtt{s}(\mathtt{add}(x, y)) & \mathtt{mult}(\mathtt{s}(x), y) &\rightarrow \mathtt{add}(y, \mathtt{mult}(x, y)) \end{aligned}$$

These functions are clearly ordered by LPO by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult}$. And they admit the quasi-interpretations:

$$\begin{aligned} (\!|\mathtt{add}|\!)(X, Y) &= X + Y \\ (\!|\mathtt{mult}|\!)(X, Y) &= X \times Y \end{aligned}$$

Remark 2. The interpretation of a polynomial correponds to its semantics.

Generalised Computability and Applications to Hybrid Systems*

Margarita V. Korovina[1] and Oleg V. Kudinov[2]

[1] A.P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Acad. Lavrentjev ave., 630090, Novosibirsk, Russia
rita@inet.ssc.nsu.ru
[2] Institute of Mathematics, Koptug Pr., 4,
Novosibirsk, Russia
kud@math.nsc.ru

Abstract. We investigate the concept of generalised computability of operators and functionals defined on the set of continuous functions, firstly introduced in [9]. By working in the reals, with equality and without equality, we study properties of generalised computable operators and functionals. Also we propose interesting applications to formalisation of hybrid systems. We obtain some class of hybrid systems, which trajectories are computable in the sense of computable analysis.

1 Introduction

Computability theories on particular and general classes of structures address central concerns in mathematics and computer science. The concept of generalised computability is closely related to definability theory investigated in [3]. This theory has many uses in computer science and mathematics because it can be applied to analyse computation on abstract structure, in particularly, on the real numbers or on the class of continuous functions. The main aim of our paper is to study properties of operators and functionals considering their generalised computability relative either to the ordered reals with equality, or to the strictly ordered real field. Note that generalised computability related to the strictly ordered real field is equivalent to computability in computable analysis in that they define the same class of computable real-valued functions and functionals. We prove that any continuous operator is generalised computable in the language with equality if and only it is generalised computable in the language without equality. As a direct corollary we obtain that each continuous generalised computable with equality operator is computable in the sense of computable analysis. This paper is structured as follows. In Section 2, we give basic definitions and tools. We study properties of operators and functionals considering their generalised computability in the language with equality and in

* This research was supported in part by the RFBR (grants N 99-01-00485, N 00-01-00810) and by the Siberian Branch of RAS (a grant for young researchers, 2000)

the language without equality. In Section 3 we present some application of the proposed model of computation to specification of hybrid systems. In the recent time, attention to the problems of exact mathematical formalisation of complex systems such as hybrid systems is constantly raised. By a hybrid system we mean a network of digital and analog devices interacting at discrete times. An important characteristic of hybrid systems is that they incorporate both continuous components, usually called plants, as well as digital components, i.e. digital computers, sensors and actuators controlled by programs. These programs are designed to select, control, and supervise the behaviour of the continuous components. Modelling, design, and investigation of behaviours of hybrid systems have recently become active areas of research in computer science (for example see [5, 10,12,13]). The main subject of our investigation is behaviour of the continuous components. In [12], the set of all possible trajectories of the plant was called as a performance specification. Based on the proposed model of computation we introduce logical formalisation of hybrid systems in which the trajectories of the continuous components (the performance specification) are presented by computable functionals.

2 Generalised Computability

Throughout the article we consider two models of the real numbers,

$$<\mathbb{R},\sigma_1> \rightleftharpoons <\mathbb{R},0,1,+,\cdot,<,-x,\frac{x}{2}>$$

is the model of the reals without equality, and

$$<\mathbb{R},\sigma_2> \rightleftharpoons <\mathbb{R},0,1,+,\cdot,\leq>$$

is the model of the reals with equality. Below if statements concern languages σ_1 and σ_2 we will write σ for a language.

Denote $\mathbf{D}_2 = \{z \cdot 2^{-n} | z \in \mathbb{Z},\ n \in \mathbb{N}\}$. Let us use $\bar{r}$ to denote $r_1, \ldots, r_m$.

To recall the notion of generalised computability, let us construct the set of hereditarily finite sets $\mathrm{HF}(M)$ over a model $\mathbf{M}$. This structure is rather well studied in the theory of admissible sets [1] and permits us to define the natural numbers and to code and store information via formulas. Let $\mathbf{M}$ be a model of a language σ whose carrier set is M. We construct the set of hereditarily finite sets, $\mathrm{HF}(M) = \bigcup_{n\in\omega} \mathrm{S}_n(M)$, where $\mathrm{S}_0(M) \rightleftharpoons M$, $\mathrm{S}_{\mathrm{n}+1}(M) \rightleftharpoons \mathcal{P}_\omega(\mathrm{S}_\mathrm{n}(M)) \cup \mathrm{S}_\mathrm{n}(M)$, where $n \in \omega$ and for every set B, $\mathcal{P}_\omega(B)$ is the set of all finite subsets of B.

We define $\mathbf{HF(M)} \rightleftharpoons \left\langle \mathrm{HF}(M), M, \sigma, \emptyset_{\mathbf{HF(M)}}, \in_{\mathbf{HF(M)}} \right\rangle$, where the unary predicate $\emptyset$ singles out the empty set and the binary predicate symbol $\in_{\mathbf{HF(M)}}$ has the set-theoretic interpretation.

Below we will consider $M \rightleftharpoons \mathbb{R}$, $\sigma_1^* = \sigma_1 \cup \{\in,\ \emptyset\}$ named the language without equality and $\sigma_2^* = \sigma_2 \cup \{\in,\ \emptyset\}$ named the language with equality. Below if statements concern both languages σ_1^* and σ_2^* we will write σ^* for a language.

To introduce the notions of terms and atomic formulas we use variables of two sorts. Variables of the first sort range over $\mathbb{R}$ and variables of the second sort range over $\mathrm{HF}(\mathbb{R})$.

The terms in the language σ_1^* are defined inductively by: 1. the constant symbols 0 and 1 are terms; 2. the variables of the first sort are terms; 3. if t_1, t_2 are terms then $t_1 + t_2$, $t_1 \cdot t_2$, $-t_1$, $\frac{t_2}{2}$ are terms. The notions of a term in the language σ_2^* can be given in a similar way.

The following formulas in the language σ_1^* are atomic: $t_1 < t_2$, $t \in s$ and $s_1 \in s_2$ where t_1, t_2, t are terms and s_1, s_2 are variables of the second sort. The following formulas in the language σ_2^* are atomic: $t_1 \leq t_2$, $t \in s$, $s_1 \in s_2$ where t_1, t_2, t are terms and s_1, s_2 are variables of the second sort.

The set of *Δ_0-formulas* in the language σ^* is the closure of the set of atomic formulas in the language σ^* under $\wedge, \vee, \neg, (\exists x \in s)$ and $(\forall x \in s)$, where $(\exists x \in s)\ \varphi$ denotes $\exists x(x \in s \wedge\ \varphi)$ and $(\forall x \in s)\ \varphi$ denotes $\forall x(x \in s \to \varphi)$ and s is any variable of second type. The language σ_1 is the language of strictly ordered real field, so the predicate $<$ occurs positively in Δ_0-formulas in the language σ_1^*.

The set of *Σ-formulas* in the language σ^* is the closure of the set of Δ_0 formulas in the language σ^* under $\wedge, \vee, (\exists x \in s), (\forall x \in s)$, and $\exists$. The natural numbers $0,\ 1, \ldots$ are identified with $\emptyset,\ \{\emptyset, \{\emptyset\}\}, \ldots$ so that, in particular, $n+1 = n \cup \{n\}$ and the set ω is a subset of HF(IR).

Definition 1. *A relation $B \subseteq \mathbb{R}^n$ is Σ-definable in σ^*, if there exists a Σ-formula $\Phi(\bar{x})$ in the language σ^* such that $\bar{x} \in B \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \Phi(\bar{x})$. A function is Σ-definable if its graph is Σ-definable.*

Note that the set $\mathbb{R}$ is Δ_0–definable in both the languages σ_1^* and σ_2^*. This fact makes $\mathbf{HF}(\mathbb{R})$ a suitable domain for studying relations in $\mathbb{R}^n$ and functions from $\mathbb{R}^n$ to $\mathbb{R}$ where $n \in \omega$. For properties of Σ-definable relations in $\mathbb{R}^n$ we refer to [3,6].
Without loss of generality we consider the set of continuous functions defined on the compact interval $[0, 1]$. To introduce generalised computability of operators and functionals we extend σ_1^* and σ_2^* by two 3-ary predicates U_1 and U_2.

Definition 2. *Let $\varphi_1(U_1, U_2, x_1, x_2, c)$, $\varphi_2(U_1, U_2, x_1, x_2, c)$ be formulas of extended language σ_1^* (σ_2^*). We suppose that U_1, U_2 occur positively in φ_1, φ_2 and the predicates U_1, U_2 define open sets in $\mathbb{R}^3$. The formulas φ_1, φ_2 are said to satisfy joint continuity property if the following formulas are valid in $\mathbf{HF}(\mathbb{R})$.*

1. $\forall x_1 \forall x_2 \forall x_3 \forall x_4 \forall z\, ((x_1 \leq x_3) \wedge (x_4 \leq x_2) \wedge \varphi_i(U_1, U_2, x_1, x_2, z)) \to$ $\varphi_i(U_1, U_2, x_3, x_4, z)$, *for* $i = 1, 2$
2. $\forall x_1 \forall x_2 \forall c \forall z\, ((z < c) \wedge \varphi_1(U_1, U_2, x_1, x_2, c)) \to \varphi_1(U_1, U_2, x_1, x_2, z)$,
3. $\forall x_1 \forall x_2 \forall c \forall z\, ((z > c) \wedge \varphi_2(U_1, U_2, x_1, x_2, c)) \to \varphi_2(U_1, U_2, x_1, x_2, z)$,
4. $\forall x_1 \forall x_2 \forall x_3 \forall z\, (\varphi_i(U_1, U_2, x_1, x_2, z) \wedge \varphi_i(U_1, U_2, x_2, x_3, z)) \to$ $\varphi_i(U_1, U_2, x_1, x_3, z)$, *for* $i = 1, 2$,
5. $(\forall y_1 \forall y_2 \exists z \forall z_1 \forall z_2 (U_1(y_1, y_2, z_1) \wedge U_2(y_1, y_2, z_1) \to (z_1 < z < z_2))) \to$ $(\forall x_1 \forall x_2 \exists c \forall c_1 \forall c_2 (\varphi_1(U_1, U_2, x_1, x_2, c_1) \wedge \varphi_2(U_1, U_2, x_1, x_2, c_2) \to$ $(c_1 < c < c_2)))$.

Definition 3. *A partial operator $F : C[0,1] \to C[0,1]$ is said to be shared by two Σ-formulas φ_1 and φ_2 in the language σ^* if the following assertions hold. If $F(u) = h$ then $h|_{[x_1,x_2]} > z \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \varphi_1(U_1, U_2, x_1, x_2, z)$ and $h|_{[x_1,x_2]} < z \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \varphi_2(U_1, U_2, x_1, x_2, z)$, where $U_1(x_1, x_2, c) \rightleftharpoons u|_{[x_1,x_2]} > c, U_2(x_1, x_2, c) \rightleftharpoons u|_{[x_1,x_2]} < c$ and U_1, U_2 occur positively in φ_1, φ_2.*

Definition 4. *A partial operator $F : C[0,1] \to C[0,1]$ is said to be generalised computable in the language σ^*, if F is shared by two Σ-formulas in the language σ^* which satisfy the joint continuity property.*

Definition 5. *A partial functional $F : C[0,1] \times [0,1] \to \mathbb{R}$ is said to be generalised computable in the language σ^*, if there exists an operator $F^* : C[0,1] \to C[0,1]$ generalised computable in the language σ^* such that $F(f,x) = F^*(f)(x)$.*

Definition 6. *A partial functional $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ is said to be generalised computable in the language σ^*, if there exists an effective sequence $\{F_n^*\}_{n\in\omega}$ of operators generalised computable in the language σ^* of the types $F_n^* : C[0,1] \to C[-n,n]$ such that $F(f,x) = y \leftrightarrow \forall n \left(-n \le x \le n \to F_n^*(f)(x) = y\right)$.*

Proposition 1. *A partial functional $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ is generalised computable in the language without equality if and only if it is computable in the sense of computable analysis.*

Proof. See [9,17].

Now we propose the main theorem which connects generalised computabilities in the various languages.

Theorem 1. *A continuous total operator $F : C[0,1] \to C[0,1]$ is generalised computable in the language with equality if and only if it is generalised computable in the language without equality.*

Proposition 2. *Let $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ be a continuous total functional. The functional F is generalised computable in the language with equality if and only if it is generalised computable in the language without equality.*

Corollary 1. *Let $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ be a continuous total functional. The functional F is generalised computable if and only if F is computable in the sense of computable analysis.*

Now we point attention to a useful recursion scheme, which permits us to describe the behaviour of complex systems such as hybrid systems.

Let $\mathcal{F} : C[0,1] \times C[0,1] \times \mathbb{R} \to \mathbb{R}$ and $G : C[0,1] \times [0,1] \to \mathbb{R}$ be generalised computable in the language with equality functionals. Then $F : C[0,1] \times [0,+\infty) \to \mathbb{R}$ is defined by the following scheme:

$$\begin{cases} F(f,t)|_{t\in[0,1]} = G(f,t), \\ F(f,t)|_{t\in(n,n+1]} = \mathcal{F}(f,t,\lambda y F(f, y+n-1)) \end{cases}$$

Proposition 3. *The functional F is generalised computable in the language with equality, with F defined above.*

3 An Application to Formalisation of Hybrid Systems

We use the models of hybrid systems proposed by Nerode, Kohn in [12]. A hybrid system is a system which consists of a continuous plant that is disturbed by external world and controlled by a program implemented on a sequential automaton. The main subject of our investigation is behaviour of the continuous components. We propose a logical formalisation of hybrid systems in which the trajectories of the continuous components (the performance specification) are presented by computable functionals.

A formalisation of the hybrid system $\mathbf{FHS} = \langle TS, \mathbf{F}, Conv1, A, Conv2, I\rangle$ consists of:

- $TS = \{t_i\}_{i\in\omega}$. It is an effective sequence of rational numbers i.e. the set of Gordel numbers of elements of TS is computable enumerated. The rational numbers t_i are the times of communication of the external world and the hybrid system, and communication of the plant and the control automaton. The time sequence $\{t_i\}_{i\in\omega}$ satisfies the realizability requirements:
  1. For every i, $t_i \geq 0$; $t_0 < t_1 < \ldots < t_i \ldots$;
  2. The differences $t_{i+1} - t_i$ have positive lower bounds.
- $\mathbf{F} : C[0,1] \times \mathrm{I\!R}^n \to \mathrm{I\!R}$. It is a generalised computable functional in the language σ_2^*. The plant has been given by this functional.
- $Conv1 : C[0,1] \times \mathrm{I\!R} \to A^*$. It is an generalised computable functional in the following sense: $Conv1(f,x) = w \Leftrightarrow \varphi(U_1, U_2, x, w)$, where $U_1(x_1,x_2,c) \rightleftharpoons f|_{[x_1,x_2]} > c, U_2(x_1,x_2,c) \rightleftharpoons f|_{[x_1,x_2]} < c$ and the predicate U_1 and U_2 occur positively in Σ-formula φ. At the time of communication this functional converts measurements, presented by the meaning of $\mathbf{F}$, and the representation of external world f into finite words which are input words of the internal control automata.
- $A : A^* \to A^*$. It is a Σ-definable function. The internal control automata, in practice, is a finite state automata with finite input and finite output alphabets. So, it is naturally modelled by Σ-definable function (see [3,6]) which has a symbolic representation of measurements as input and produces a symbolic representation of the next control law as output.
- $Conv2 : A^* \to \mathrm{I\!R}^{n-1}$. It is a Σ-definable function. This function converts finite words representing control laws into control laws imposed on the plant.
- $I \subset A^* \cup \mathrm{I\!R}^n$. It is a finite set of initial conditions.

Definition 7. *The behaviour of a hybrid system is defined by a functional* $H : C[0,1] \times \mathrm{I\!R} \to \mathrm{I\!R}$ *if for any external disturbation* $f \in C[0,1]$ *the values of* $H(f,\cdot)$ *define the trajectory of the hybrid system.*

Theorem 2. *Suppose a hybrid system is specified as above. If the behaviour of the hybrid system is defined by a continuous functional* H *then* H *is computable in the sense of computable analysis.*

Proof. The claim follows from Theorem 1 and Proposition 3. □

In conclusion we would like to note that subjects of future papers will be formulation and investigation of optimal hybrid control in terms of generalised computability.

References

1. J. Barwise, Admissible sets and structures, Berlin, Springer–Verlag, 1975.
2. A. Edalat, P. Sünderhauf, A domain-theoretic approach to computability on the real line, Theoretical Computer Science, 210, 1998, pages 73–98.
3. Yu. L. Ershov, Definability and computability, Plenum, New York, 1996.
4. A. Grzegorczyk, On the definitions of computable real continuous functions, Fund. Math., N 44, 1957, pages 61–71.
5. T.A. Henzinger, Z. Manna, A. Pnueli, Towards refining Temporal Specifications into Hybrid Systems, LNCS N 736, 1993, pages 36–60.
6. M. Korovina, O. Kudinov, A New Approach to Computability over the Reals, SibAM, v. 8, N 3, 1998, pages 59–73.
7. M. Korovina, O. Kudinov, Characteristic Properties of Majorant-Computability over the Reals, Proc. of CSL'98, LNCS, 1584, 1999, pages 188–204.
8. M. Korovina, O. Kudinov, A Logical approach to Specifications of Hybrid Systems, Proc. of PSI'99, LNCS 1755, 2000, pages 10–16.
9. M. Korovina, O. Kudinov, Formalisation of Computability of Operators and Real-Valued Functionals via Domain Theory, Proceedings of CCA-2000, to appear in LNCS, 2001.
10. Z. Manna, A. Pnueli, Verifying Hybrid Systems, LNCS N 736, 1993, pages 4–36.
11. R. Montague, Recursion theory as a branch of model theory, Proc. of the third international congr. on Logic, Methodology and the Philos. of Sc., 1967, Amsterdam, 1968, pages 63–86.
12. A. Nerode, W. Kohn, Models for Hybrid Systems, Automata, Topologies, Controllability, Observability, LNCS N 736, 1993, pages 317–357.
13. W. Kohn, A. Nerode, J. B. Remmel Agent Based Velocity Control of Highway Systems, LNCS N 1273, 1997, pages 174–215.
14. M. B. Pour-El, J. I. Richards, Computability in Analysis and Physics, Springer-Verlag, 1988.
15. D. Scott, Outline of a mathematical theory of computation, In 4th Annual Princeton Conference on Information Sciences and Systems, 1970, pages 169–176.
16. Viggo Stoltenberg-Hansen, John V. Tucker, Concrete models of computation for topological algebras, TCS 219, 1999, pages 347-378.
17. K. Weihrauch, Computable analysis. An introduction, Springer, 2000.

Exploring Template Template Parameters

Roland Weiss and Volker Simonis

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{weissr,simonis}@informatik.uni-tuebingen.de

Abstract. The generic programming paradigm has exerted great influence on the recent development of C++, e.g., large parts of its standard library [2] are based on generic containers and algorithms. While templates, the language feature of C++ that supports generic programming, have become widely used and well understood in the last years, one aspect of templates has been mostly ignored: template template parameters ([2], 14.1). In the first part, this article will present an in depth introduction of the new technique. The second part introduces a class for arbitrary precision arithmetic, whose design is based on template template parameters. Finally, we end with a discussion of the benefits and drawbacks of this new programming technique and how it applies to generic languages other than C++.

1 Introduction

The C++ standard library incorporated the standard template library (STL) [15] and its ideas, which are the cornerstones of generic programming [14]. Templates are the language feature that supports generic programming in C++. They come in two flavors, class templates and function templates. Class templates are used to express classes parameterized with types, e.g., the standard library containers, which hold elements of the argument type. Generic algorithms can be expressed with function templates. They allow one to formulate an algorithm independently of concrete types, such that the algorithm is applicable to a range of types complying to specific requirements. For example, the standard `sort` algorithm without function object ([2], 25.3) is able to rearrange a sequence of arbitrary type according to the order implied by the comparison operator `<`. Of course, the availability of this operator is a requirement on the elements' type.

It is possible to use instantiated class templates as arguments for class and function templates, therefore one is able to write nested constructs like `vector<list<long> >`. So where does the need for template template parameters arise? Templates give one the power to abstract from an implementation detail, the types of the application's local data. Template template parameters provide one with the means to introduce an additional level of abstraction. Instead of using an instantiated class template as argument, the class template itself can be used as template argument. To clarify the meaning of this statement, we will look in the following sections at class and function templates that take template template parameters. Then we will present a generic arbitrary precision arithmetic implemented with template template parameters. Finally, the presented technique is discussed and effects on other generic languages are considered.

2 Class Templates

The standard library offers three sequence containers, `vector`, `list` and `deque`. They all have characteristics that recommend them for a given application context. But if one wants to write a new class called `store` that uses a standard container internally to store values, it is hard to choose the *perfect* container for all possible scenarios. This is exactly the situation where template template parameters fit in. The class designer can provide a default container, but the user can override this decision easily. Note that the user can not only use standard containers but also any proprietary container that conforms to the standard sequence container interface. Let us look at a code example that implements the class `store_comp` using object composition.

```
template < typename val_t,
  template <typename T, typename A> class cont_t = std::deque,
  typename alloc_t = std::allocator<val_t> >
class store_comp
{
  cont_t<val_t, alloc_t> m_cont; //instantiate template template parameter
public:
  typedef typename cont_t<val_t, alloc_t<val_t> >::iterator iterator;
  iterator begin() { return m_cont.begin(); }
  // more delegated methods...
};
```

The first template parameter `val_t` is the type of the objects to be kept inside the store. `cont_t`, the second one, is the template template parameter, which we are interested in. The declaration states that `cont_t` expects two template parameters `T` and `A`, therefore any standard conforming sequence container is applicable. We also provide a default value for the template template parameter, the standard container `deque`. When working with template template parameters, one has to get used to the fact that one provides a real class template as template argument, not an instantiation. The container's allocator `alloc_t` defaults to the standard allocator.

There is nothing unusual about the usage of `cont_t`, the private member `m_cont` is an instantiation of the default or user provided sequence container. As already mentioned, this implementation of `store_comp` applies composition to express the relationship between the new class and the internally used container. Another way to reach the same goal is to use inheritance, as shown in the following code segment:

```
template <typename val_t, ...>
class store_inh : public cont_t<val_t, alloc_t<val_t> > {};
```

The template header is the same as in the previous example. Due to the public inheritance, the user can work with the container's typical interface to change the store's content. For the class `store_comp`, appropriate member functions must be written, which delegate the actual work to the private member `m_cont`. The two differing designs of class `store` are summarized in Figure 1. The notation follows the diagrams in [9]. The only extension is that template template parameters inside the class' parameter list are typeset in boldface.

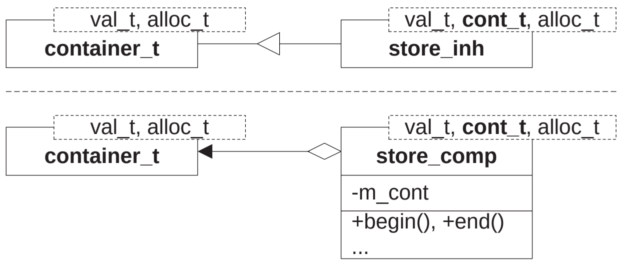


Fig. 1. Comparison of the competing designs of the `store` classes

To conclude the overview, these code lines show how to create instances of the `store` classes:

```
store_comp<std::string, std::list> sc;
store_inh<int> si;
```

`sc` uses a `std::list` as internal container, whereas `si` uses the default container `std::deque`. This is a very convenient way for the user to select the appropriate container that matches the needs in his application area. The template template parameter can be seen as a container *policy* [1].

Now that we have seen how to apply template template parameters to a parameterized class in general, let us examine some of the subtleties.

First, the template template parameter – `cont_t` in our case – must be introduced with the keyword `class`, `typename` is not allowed ([2], 14.1). This makes sense, since a template template argument must correspond to a class template, not just a simple type name.

Also, the identifiers `T` and `A` introduced in the parameter list of the template template parameter are only valid inside its own declaration. Effectively, this means that they are not available inside the scope of the class `store`. One can instantiate the template template parameter inside the class body with different arguments multiple times, which would render the identifier(s) ambiguous. Hence, this scoping rule is reasonable.

But the most important point is the number of parameters of the template template parameter itself. Some of you may have wondered why two type parameters are given for a standard container, because they are almost exclusively instantiated with just the element type as argument, e.g., `std::deque<float>`.

In these cases, the allocator parameter defaults to the standard allocator. Why do we have to declare it for cont_t? The answer is obvious: the template parameter signatures of the following two class templates C1 and C2 are distinct, though some of their instantiations can look the same:

```
template <typename T> class C1 {};
template <typename T1, typename T2 = int> class C2 {};
C1<double> c1; // c1 has signature C1<double>
C2<double> c2; // c2 has signature C2<double, int>
```

In order to be able to use standard containers, we have to declare cont_t conforming to the standard library. There ([2], 23.2), all sequence containers have two template parameters.[1] This can have some unexpected consequences. Think of a library implementor who decides to add another default parameter to a sequence container. Normal usage of this container is not affected by this implementation detail, but the class store can not be instantiated with this container because of the differing number of template parameters. We have encountered this particular problem with the deque implementation of the SGI STL [23].[2] Please note that some of the compilers that currently support template template parameters fail to check the number of arguments given to a template template parameter instantiation.

The template parameters of a template template parameter can have default arguments themselves. For example, if one is not interested in parameterizing a container by its allocator, one can provide the standard allocator as default argument and instantiate the container with just the contained type.

Finally, we will compare the approach with template template parameters to the traditional one using class arguments with template parameters. Such a class would look more or less like this:

```
template <typename cont_t>
class store_t
{
  cont_t m_cont; // use instantiated container for internal representation
public:
  typedef typename cont_t::iterator iterator;            // iterator type
  typedef typename cont_t::value_type value_type;         // value type
  typedef typename cont_t::allocator_type allocator_type; // alloc type
  // rest analogous to store_comp ...
};
typedef std::list<int> my_cont; // container for internal representation
store_t<my_cont> st;            // instantiate store
```

We will examine the advantages and drawbacks of each approach. The traditional one provides an instantiated class template as template argument. Therefore, store_t can extract all necessary types like the allocator, iterator etc. This is not possible in classes with template template parameters, because they perform the instantiation of the internal container themselves.

[1] The C++ Standardization Committee currently discusses if this a defect, inadequately restricting library writers.

[2] The additional third template parameter was removed recently.

But the traditional approach was made applicable at all by the fact that the user provides the type with which the sequence container is instantiated. If the type is an implementation detail not made explicit to the user, the traditional approach doesn't work. See [21] for an application example with these properties. The ability to create multiple, different instantiations inside the class template body using the template template argument is also beyond the traditional approach:

```
cont_t<int, alloc_t> cont_1;
cont_t<val_t, std::allocator<val_t> > cont_2;
```

3 Function Templates

In the preceding section we showed that by application of template template parameters we gain flexibility in building data structures on top of existing STL container class templates. Now we want to examine what kind of abstractions are possible for generic functions with template template parameters. Of course, one can still use template template parameters to specify a class template for internal usage. This is analogous to the class `store_comp`, where object composition is employed.

But let us try to apply a corresponding abstraction to generic functions as we did to generic containers. We were able to give class users a convenient way to customize a complex data structure according to their application contexts. Transferring this abstraction to generic functions, we want to provide functions whose behavior is modifiable by their template template arguments.

We will exemplify this by adding a new method `view` to the class `store`. Its purpose is to print the store's content in a customizable way. A bare bones implementation inside a class definition is presented here:

```
template <template <typename iter_t> class mutator>
void view(std::ostream& os)
{
  mutator<iterator>()(begin(),end()); // iterator: defined in the store
  std::copy(begin(), end(), std::ostream_iterator<val_t>(os, " "));
}
```

Here, `mutator` is the template template parameter, it has an iterator type as template parameter. The `mutator` changes the order of the elements that are delimited by the two iterator arguments and then prints the changed sequence. This behavior is expressed in the two code lines inside the method body. The first line instantiates the `mutator` with the store's iterator and invokes the `mutator`'s application operator, where the elements are rearranged. In the second line, the mutated store is written to the given output stream `os`, using the algorithm `copy` from the standard library. The types `iterator` and `val_t` are defined in the store class.

The first noteworthy point is that we have to get around an inherent problem of C++: functions are not first order objects. Fortunately, the same workaround already applied to this problem in the STL works fine. The solution is to use function objects (see [15], chapter 8). In the `view` method above, a function object that takes two iterators as arguments is required.

The following example shows how to write a function object that encapsulates the `random_shuffle` standard algorithm and how to call `view` with this function object as the `mutator`:

```
// function object that encapsulates std::random_shuffle
template <typename iter_t>
struct RandomShuffle
{
  void operator()(iter_t i1, iter_t i2) { std::random_shuffle(i1, i2); }
};
// A store s must be created and filled with values...
s.view<RandomShuffle>(cout); //RandomShuffle is the mutator
```

There are two requirements on the template arguments such that the presented technique works properly. First, the application operator provided by the function object, e.g., `RandomShuffle`, must match the usage inside the instantiated class template, e.g., `store_comp`. The `view` method works fine with application operators that expect two iterators as input arguments, like the wrapped `random_shuffle` algorithm from the standard library.

The second requirement touches the generic concepts on which the STL is built. `RandomShuffle` wraps the `random_shuffle` algorithm, which is specified to work with random access iterators. But what happens if one instantiates the `store` class template with `std::list` as template template argument and calls `view<RandomShuffle>`? `std::list` supports only bidirectional iterators, therefore the C++ compiler must fail instantiating `view<RandomShuffle>`. If one is interested in a function object that is usable with all possible `store` instantiations, two possibilities exist. Either we write a general algorithm and demand only the weakest iterator category, possibly loosing efficiency. Or we apply a technique already used in the standard library. The function object can have different specializations, which dispatch to the most efficient algorithm based on the iterator category. See [4] for a good discussion of this approach. This point, involving iterator and container categories as well as algorithm requirements, emphasizes the position of Musser et. al. [16] that generic programming is requirement oriented programming.

Completing, we want to explain why template template parameters are necessary for the `view` function and simple template parameters won't suffice. The key point is that the `mutator` can only be instantiated with the correct iterator. But the iterator is only know to the `store`, therefore an instantiation outside the class template `store` is not possible, at least not in a consistent manner.

Overall, the presented technique gives a class or library designer a versatile tool to make functions customizable by the user.

4 Long Integer Arithmetic – An Application Example

Now we will show how the techniques introduced in the last two sections can be applied to a real world problem. Suppose you want to implement a library for arbitrary precision arithmetic. One of the main problems one encounters is the question of how to represent long numbers. There are many well known possibilities to choose from: arrays, single linked lists, double linked lists, garbage collected or dynamically allocated and freed storage and so on. It is hard to make the right decision at the beginning of the project, especially because our decision will influence the way we have to implement the algorithms working on long numbers. Furthermore, we might not even know in advance all the algorithms that we eventually want to implement in the future.

The better way to go is to leave this decision open and parameterize the long number class by the container, which holds the digits. We just specify a minimal interface where every long number is a sequence of digits, and the digits of every sequence have to be accessible through iterators. With this in mind, we can define our long number class as follows:

```
template<
  template<typename T, typename A> class cont_t = std::vector,
  template<typename AllocT> class alloc_t = std::allocator
>
class Integer {
  // ..
};
```

The first template template parameter stands for an arbitrary container type, which fulfills the requirements of a STL container. As we do not want to leave the memory management completely in the container's responsibility, we use a second template template parameter, which has the same interface as the standard allocator. Both template template parameters have default parameters, namely the standard vector class `std::vector` for the container and the standard allocator std::allocator for the allocator.

Knowing only this interface, a user could create `Integer` instances, which use different containers and allocators to manage a long number's digits. He even does not have to know if we use composition or inheritance in our implementation (see Figure 1 for a summary of the two design paradigms).[3]

In order to give the user access to the long number's digits, we implement the methods `begin()`, `end()` and `push_back()`, which are merely wrappers to the very same methods of the parameterized container. The first two return iterators that give access to the actual digits while the last one can be used to append a digit at the end of the long number. Notice that the type of a digit is treated as an implementation detail. We only have to make it available by defining a public type called `digit_type` in our class. Also we hand over in this way the type definitions of the iterators of the underlying containers. Now, our augmented class looks as follows (with the template definition omitted):

[3] We used composition in our implementation. The main reason was that we wanted to minimize the tradeoff between long numbers consisting of just one digit and real long numbers. Therefore, our `Integer` class is in fact a kind of union or variant record in Pascal notation of either a pointer to the parameterized container or a plain digit. The source code of our implementation is available at http://www-ca.informatik.uni-tuebingen.de/people/simonis/projects.htm.

```
class Integer {
public:
  typedef int digit_type;
  typedef typename cont_t::iterator iterator;
  iterator begin() { return cont->begin(); }
  iterator end() { return cont->end(); }
  void push_back(digit_type v) { cont->push_back(v); }
private:
  cont_t<digit_type, alloc_t> *cont;
};
```

With this in mind and provided addition is defined for the digit type, a user may implement a naive addition without carry for long numbers of equal length in the following way (again the template definition has been omitted):

```
Integer<cont_t, alloc_t>
add(Integer<cont_t, alloc_t> &a, Integer<cont_t, alloc_t> &b) {
  Integer<cont_t, alloc_t> result;
  typename Integer<cont_t, alloc_t>::iterator ia=a.begin(), ib=b.begin();
  while(ia != a.end()) result.push_back(*ia + *ib);)
  return result;
}
```

Based on the technique of iterator traits described in [5] and the proposed container traits in [4] specialized versions of certain algorithms may be written, which make use of the specific features of the underlying container. For example, an algorithm working on vectors can take advantage of random access iterators, while at the same time being aware of the fact that insert operations are linear in the length of the container.

5 Conclusions and Perspectives

We have shown how template template parameters are typically employed. They can be used to give library and class designers new power in providing the user with a facility to adapt the predefined behavior of classes and functions according to his needs and application context. This is especially important if one wants to build on top of already existing generic libraries like the STL.

With our example we demonstrate how template template parameters and generic programming can be used to achieve a flexible design. In contrast to usual template parameters, which parameterize with concrete types, template template parameters allows one to parameterize with incomplete types. This is a kind of *structural* abstraction compared to the abstraction over simple types achieved with usual template parameters. As templates are always instantiated at compile time, this technique comes with absolutely no runtime overhead compared to versions which don't offer this type of parameterization.

One has to think about the applicability of template template parameters, a C++ feature, to other programming languages. Generally, a similar feature makes sense in every language that follows C++'s instantiation model of resolving all type bindings at compile time (e.g., Modula-3 and Ada). Template

Table 1. Performance comparison of our Integer class compared to other arbitrary precision libraries. While GMP is a C library with optimized assembler routines, all the other libraries are written in C++. The first line of every entry denotes the number of processor instructions while the second one indicates the number of processor cycles needed for one operation. Integer† stands for `Integer<slist>`, Integer‡ for `Integer<std::list>`, and Integer§ for `Integer<std::vector>`. The “RW-” prefix marks tests, which have been taken with the Rogue Wave STL in contrast to the other tests, which used the SGI STL

bits	GMP	CLN	NTL	Piologie	Integer†	Integer‡	RW-Integer‡	Integer§	RW-Integer§
Addition of n-Bit numbers									
128	820	718	2.087	509	3.693	4.119	3.275	1.658	2.038
	3.846	4.375	6.080	3.290	6.186	6.723	7.186	4.411	4.014
1.024	1.001	871	2.649	1.210	16.769	18.025	7.671	3.078	2.974
	4.336	4.596	6.350	5.392	17.512	17.961	10.892	5.574	4.497
8.192	1.691	1.971	7.350	3.748	121.805	128.774	40.046	13.456	11.668
	5.317	7.986	12.603	8.601	104.750	113.095	45.748	14.231	11.241
65.536	8.174	10.505	43.530	23.491	960.955	1.015.816	278890	96.657	80.150
	32.217	48.811	65.176	52.254	844.618	952.695	339371	104.143	77.745
Multiplication of n-Bit numbers									
128	922	990	1.900	1.293	6.181	6.687	4.088	2.798	2.646
	6.450	4.513	6.972	5.461	11.744	12.674	11.601	9.682	6.743
1.024	8.815	13.740	28.837	34.383	71.585	72.311	52.074	40.234	32.564
	16.572	24.736	29.671	43.539	60.661	63.933	48.949	35.594	30.221
8.192	240.738	394.093	828.825	940.873	2.852.491	2.786.261	2.749.538	2.399.391	1.882.072
	243.438	523.903	671.630	926.693	1.853.809	1.971.483	18.67.466	1.715.800	1.488.350
65.536	5.477.327	13.370.805	22.798.590	28.939.305	167.792.489	163.149.346	171.090.108	151.754.471	118.611.246
	5.158.666	14.695.137	18.031.103	27.289.599	117.485.455	122.771.953	120.008.350	107.798.237	93.442.537

template parameters are a powerful feature to remove some restrictions imposed by such a strict instantiation model without introducing runtime overhead.

We measured our example with GCC 2.97 and two versions of the STL, namely the from SGI [23] and one from Rogue Wave [20]. Table 1 compares our `Integer` class with some widely available arbitrary precision libraries (GMP 3.1.1 [11], CLN 1.0.3 [12], NTL 4.1a [22] and Piologie 1.2.1 [24]). The tests have been done on a PentiumIII 667MHz Linux system using the PCL library [6].

The results of some tests with garbage collected containers using the Boehm-Weiser-Demers [7] collector have been not very promising. However the significant performance difference between the two STL versions we used indicate that this may be no fundamental problem, but a problem of bad compiler optimization and the orthogonal design of the SGI-STL containers and the plain Boehm-Weiser-Demers garbage collector. Therefor we plan further tests in the future using optimizing compiler and other collectors like TGC [18], [19], which address exactly this problems.

6 Compiler Support

One major problem in working with template template parameters is not a conceptual, but rather a practical one. Even now, three years after the publication of the ISO C++ standard, not all compilers implement this feature.

We were able to compile our examples only with the following compilers: Borland C++ V5.5 [3], Visual Age C++ V4.0 [13], Metrowerks V6.0 and all compilers based on the edg front-end V2.43 [8]. The snapshot versions after November 2000 of the Gnu C++ Compiler [10] also meet the requirements.

Acknowledgements. We want to thank Rüdiger Loos for his ideas on nested lists and long integers based on STL containers, which sparked our interest in template template parameters. We also want to thank the guys from EDG [8] for always supplying us with the newest version of their fabulous compiler and Martin Sebor from Rogue Wave for making available to us the latest version of their STL.

References

1. Andrei Alexandrescu. *Modern C++ Design*, Addison-Wesley Publishing Company, 2001.
2. ANSI/ISO Standard. *Programming languages - C++*, ISO/IEC 14882, 1998.
3. Borland, Inprise Corporation. *Borland C++ Compiler 5.5*, www.borland.com/bcppbuilder/freecompiler.
4. Matthew Austern. *Algorithms and Containers*, C++ Report, p. 44-47, 101 communications, July/August 2000.
5. Matthew Austern. *Generic Programming and the STL*, Addison-Wesley Publishing Company, 1999.
6. Rudolf Berrendorf, Bernd Mohr. *PCL - The Performance Counter Library*, www.fz-juelich.de/zam/PCL.

7. Hans Boehm. *Boehm-Weiser-Demers collector*, www.hpl.hp.com/personal/Hans_Boehm/gc.
8. Edison Design Group. *The C++ Front End*, www.edg.com/cpp.html.
9. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*, Addison-Wesley Publishing Company, 1995.
10. GCC steering committee. *GCC Home Page*, gcc.gnu.org.
11. Torbjorn Granlund. *GNU MP - The GNU Multi Precision Library*, www.swox.com/gmp.
12. Bruno Haible. *CLN*, clisp.cons.org/~haible/packages-cln.html.
13. IBM. *IBM VisualAge C++*, www.software.ibm.com/ad/visualage_c++.
14. Mehdi Jazayeri, Rüdiger G. K. Loos, David R. Musser (Eds.). *Generic Programming*, LNCS No. 1766, Springer, 2000.
15. David R. Musser, Gillmer J. Derge, Atul Saini. *STL Tutorial and Reference Guide: Second Edition*, Addison-Wesley Publishing Company, 2001.
16. David R. Musser, S. Schupp, Rüdiger Loos. *Requirement Oriented Programming*, in [14], p. 12-24, 2000.
17. Scott Meyers. *Effective C++ CD*, Addison-Wesley Publishing Company, 1999.
18. Gor V. Nishanov, Sibylle Schupp. *Garbage Collection in Generic Libraries*, Proc. of the ISMM 1998, p. 86-97, Richard Jones (editor), 1998.
19. Gor V. Nishanov, Sibylle Schupp. *Design and implementation of the fgc garbage collector*, Technical Report98-7, RPI, Troy, 1998.
20. Rogue Wave Software. *Rogue Wave STL (development snapshot)*, November 2000.
21. Volker Simonis, Roland Weiss. *Heterogeneous, Nested STL Containers in C++*, LNCS No. 1755 (PSI '99): p. 263-267, Springer, 1999.
22. Victor Shoup. *NTL*, www.shoup.net/ntl.
23. STL Team at SGI. *Standard Template Library Programmer's Guide*, www.sgi.com/Technology/STL.
24. Sebastian Wedeniwski. *Piologie*, ftp://ftp.informatik.uni-tuebingen.de/pub/CA/software/Piologie.

Compiler-Cooperative Memory Management in Java

Vitaly V. Mikheev and Stanislav A. Fedoseev

A.P. Ershov Institute of Informatics Systems,
Russian Academy of Sciences, Siberian Branch
Excelsior, LLC
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia
{vmikheev,sfedoseev}@excelsior-usa.com

Abstract. Dynamic memory management is a known performance bottleneck of Java applications. The problem arises out of the Java memory model in which all objects (non-primitive type instances) are allocated on the heap and reclaimed by garbage collector when they are no longer needed. This paper presents a simple and fast algorithm for inference of object lifetimes. Given the analysis results, a Java compiler is able to generate faster code, reducing the performance overhead. Besides, the obtained information may be then used by garbage collector to perform more effective resource clean-up. Thus, we consider this technique as "compile-time garbage collection" in Java.

Keywords: Java, escape analysis, garbage collection, finalization, performance

1 Introduction

Java and other object-oriented programming languages with garbage collection are widely recognized as a mainstream in the modern programming world. They allow programmers to embody problem domain concepts in a natural coding manner without paying attention to low-level implementaion details. The other side of the coin is often a poor performance of applications written in the languages. The problem has challenged compiler and run-time environment designers to propose more effective architectural decisions to reach an acceptable performance level.

A known disadvantage of Java applications is exhaustive dynamic memory consumption. For the lack of stack objects — class instances put on the stack frame, all objects have to be allocated on the heap by the *new* operator. Presence of object-oriented class libraries makes the situation much worse because any service provided by some class, prerequires the respective object allocation. Another problem inherent to Java is a so-called *pending object reclamation* [1] that does not allow garbage collector to immediately utilize some objects even though they were detected as unreachable and finalized. The Java Language Specification imposes the restriction on an implementation due to the latent caveat: if an object has a non-trivial finalizer (the **Object.finalize()** method overriden) to perform some post-mortem clean-up, the finalizer can resurrect its object "from the dead", just storing it, for instance, to a static field. Pending object reclamation reduces memory resources available to a running application.

Generally, performance issues can be addressed in either compiler or run-time environment. Most Java implementations (e.g. [2] [3]) tend to improve memory management by implementing more sophisticated algorithms for garbage collection [4]. We strongly believe that the mentioned problems should be covered in both compile-time analysis and garbage collection to use all possible opportunities for performance enhancement.

Proposition 1. *Not to junk too much is better than to permanently collect garbage*

We propose a scalable algorithm for object lifetime analysis that can be used in production compilers. We implemented the system in **JET**, Java to native code compiler and run-time environment based on the Excelsior's compiler construction framework [5].

The rest of the paper is organized as follows: Section 2 describes the program analysis and transformation for allocating objects on the stack rather than on the heap, Section 3 describes our improvements of the Java finalization mechanism. The obtained results are presented in Section 4, Section 5 highlights related works and, finally, Section 6 summarizes the paper.

2 Stack Allocating Objects

In Java programs, the lifetimes of some objects are often obvious whereas the lifetimes of others are more uncertain. Consider a simple method, getting the current date:

```
int foo() {
  Date d  = new Date();
  return d.getDate();
}
```

At the first glance, the lifetime of the object *d* is resctricted to that of method *foo*'s stack frame. That is an opportunity for a compiler to remove the *new* operator and allocate the object on the stack. However, we have to guarantee that no *d* aliases *escape* from the stack frame, that is, no aliased references to *d* are stored anywhere else. Otherwise, such program transformation would not preserve the original Java semantics. In the above example, the method *getDate* is a possible "escape direction".

Escape analysis dating back to the middle 1970s [6], addresses the problem. Many algorithms proposed vary in their application domains and time and spatial complexity. We designed a simple and fast version of escape analysis specially adapted to Java. Despite its simplicity, the algorithm shows promising results of benchmarking against widespread Java applications.

2.1 Definitions

All variables and formal parameters in the below definitions are supposed to be of Java reference types. By definition, formal parameters of a method also include the implicit "this" parameter (method receiver).

Definition 1 (Alias). *An expression expr is an alias of a variable v at a particular execution point, if v == expr (both v and expr refer to the same Java object)*

Definition 2 (Safe method). *A method is safe w.r.t its formal parameter, if any call to the method does not create new aliases for the parameter except, may be, a return value*

Definition 3 (Safe variable). *A local frame variable is safe, if no its aliases are available after method exit*

Definition 4 (Stackable type). *A reference type is stackable, if it has only a trivial finalizer*

Definition 5 (A-stackable variable). *A safe variable v is A-stackable, if a definition of v has the form of v = new T(..) for some stackable type*

Definition 6 (Stackable variable). *An A-stackable variable is stackable, if no local aliases of the variable exist before a repetitive execution of the variable definition in a loop, if any*

The stackable type definition is used to hinder a possible reference escape during finalizer invocation. The A-stackable to stackable variable refinement is introduced to preserve the semantics of the *new* operator: being executed in a loop, it creates different class instances so the analysis has to guarantee that previously cretead instances are unavailable.

2.2 Program Analysis and Transformation

To detect if a variable is not safe, we distinguish two cases of escape:

1. *explicit*: **return v**, **throw v** or **w.field = v** (an assignment to a static or instance field)
2. *implicit*: **foo (...,v, ..)** invocation of a method non-safe w.r.t **v**

Operators like **v = v1** are subject for a flow-insensitive analysis of local reference aliases (LRA) [10]. In order to meet the requirement for loop-carried variable definitions, the algorithm performs a separate LRA-analysis within loop body. Determining of safe methods is proceeded recursively as a detection of their formal parameter safety except the *return* operator. In such case, the return argument becomes involved into local reference aliasing of the calling method. We implemented our algorithm as a backward inter-procedural static analysis on call graph like algorithms described in related works [7],[9]. We omit the common analysis scheme due to its similarity to those of related works and focus on some important differencies further.

Once stackable variables have been detected, the respective v = *new T(..)* operators are replaced with the v = *stacknew T(..)* ones from internal program representation. Besides, the operators like v = *new T[expr]*, allocating variable length arrays are marked with a tag provided for subseqent code generation. That makes sense because our compiler is able to produce code for run-time stack allocation.

2.3 Implementation Notes

The Excelsior's compiler construction framework features a statistics back-end component [5] making it a suitable tool of statistic gathering and processing for any supported input language. Also, we had a memory allocation profiler in the run-time component so we were able to analyze a number of Java applications. We found that the algorithms described in related works may be somewhat simplified without sacrificing effectiveness. Moreover, the simplification often leads to better charateristics such as compilation time and resulting code size.

Type inference. So far, we have (implicitly) supposed that all called methods are available for analysis. However, Java being an object-oriented language, supports virtual method invocation — run-time method dispatching via Virtual Method Tables that hinders any static analysis. Type inference [13] is often used to avoid the problem to some extent. Our algorithm employs a context-sensitive local type inference: it starts from the known local types sourcing from local *new T(..)* operators and propagates the type information to called method context. We used a modified version of the rapid type inference pursued in [12]. Another opportunity which helps to bypass the virtual method problem is global type inference based on the class hierarchy analysis [11]. We implemented a similar algorithm but its applicability is often restricted because of the Java dynamic class loading. We did not consider polyvariant type inference (analysis of different branches at polymorphic call sites) due to its little profit in exchange for the exponential complexity.

Inline substitution. Local analysis in optimizing compilers is traditionally stronger than inter-procedural because, as a rule, it requires less resources. This is why inline substitution not only removes call overhead but also often improves code optimization. Escape analysis is not an exception from the rule: local variables that were not stackable in the called method may become so in the calling one, for instance, if references to them escaped via the *return* operator. Escape analysis in Marmot [7] specially treats called methods having that property to allocate stack variables on the frame of calling method. In the case, called method should be duplicated and specialized to add an extra reference parameter (Java supports metaprogramming so the original method signature may not be changed). In our opinion, that complicates analysis with no profit: the same problem may be solved by an ordinary inline substitution without the unnecessary code growth.

Native method models. The Java language supports external functions called *native methods*. They are usually written in C and unavailable for static analysis. However, certain native methods are provided in standard Java classes and should be implemented in any Java run-time or even compiler, for instance the **System.arraycopy** method. Because the behaviour of such methods is strictly defined by the Java Language Specification [1], we benefit from using so-called *model methods* provided for analysis purposes only. A model native method has a fake implementation simulating the original behaviour interesting for analysis. Employing model methods improves the overall precision of escape analysis.

2.4 Complexity

In according to [14], given restrictions even weaker than ours, escape analysis can be solved in linear time. The rejection of analyzing polyvariant cases at virtual call sites and the restriction of reference aliasing to local scopes only give the complexity proportional to N (program size) + G (non-virtual call graph size). Thus, our algorithm performs in O(N+G) both time and space.

3 Finalization

The described algorithm determining *safe methods* may be used for more effective implementation of pending object reclamation in Java. As mentioned above, an object having a non-trivial finalizer is prevented from immediate discarding by a garbage collector. The main problem provoking a significant memory overhead is that all heap subgraph reachable from the object may not be reclaimed as well: finalizer may potentially "save" (via aliasing) any object from the subgraph.

To overcome the drawback, we adapted the algorithm to detect whether the finalizer is a safe method with respect to its implicit "this" parameter and other object's fields aliased from "this". The analysis results are then stored by compiler to the class object (a Java metatype instance [1]). Given that, garbage collector makes a special treatment for objects with trivial or safe finalizers. More specifically, the run-time system constructs a separate list for objects which require pending reclamation whereas other objects are processed in a simpler way. The measurement results for the optimization are listed in the next section.

4 Results

We implemented the described optimizations as a part of the JET compiler and run-time environment. We selected the Javacc parser generator, the Javac bytecode compiler from Sun SDK 1.3 and Caffein Dhrystone/Strings benchmarks to evaluate resulting performance of the escape analysis application. The results are shown in Table 1 (the numbers were computed as *NewExecutionTime/OldExecutionTime*). The performance growth is achieved as a result of both faster object allocation and less extensive garbage collection.

These tests were choosen due to their batch nature that allows us to measure the difference in total execution time. Despite the results for the first three benchmarks are valuable, applying the optimization to the Javac compiler had only minimal effect — no silver bullet. Unfortunately, the results may not be directly compared with the results obtained by other researchers. The comparison of different algorithms may be accomplished only within the same optimization and run-time framework. For instance, a system with slower object allocation and garbage collection or better code optimization would obviously experience more significant performance improvement from the stack allocation.

Results of optimized finalization are given in Table 2. JFC samples (Rotator3D, Clipping, Transform, Lines) using Java 2D-graphics packages were chosen because of very intensive memory consumption. We measured the amount of free memory just after garbage collecting and the numbers were computed as *NewFreeMemory/OldFreeMemory*. The total amount of heap memory was the same for all tests and equal to 30MB.

Table 1. Stack allocating objects

Benchmark	**Execution time fraction**
Javacc	0.54
Dhrystone	0.32
Strings	0.2
Javac	0.98

Table 2. Optimized finalization

Benchmark	**Free memory fraction**	**Memory profit, MB**
Rotator3D	1.1	+1.5
Clipping	1.15	+1.2
Transform	1.08	+0.7
Lines	1.13	+1.7

We noted that even with the optimizations enabled, the total compilation time remains virtually unchanged. Analyzing obtained results, we draw a conclusion that the considered object-oriented optimizations may be employed by production compilers. All further information related to the JET project may be found at [18].

5 Related Works

An number of approaches have been proposed for object lifetime analysis. Many works were dedicated to functional languages such as SML, Lisp etc. ([14], [15], [16]). The power of the escape analyses supercedes ours to a great extent, however the complexity of the algorithms is not better than polynomial. The escape analysis for Java was investigated by reseachers using static Java analyzing frameworks. Except the JET compiler, the related works were completed on the base of the TurboJ via-C translator [9], the IBM HPJ compiler [10] and the Marmot compiler project at Microsoft Research [7]. The algorithm presented is simpler but, nevertheless, quite effective and precise so it may be used even in dynamic compilers built in the most current Java Virtual Machines [2], [3]. Besides, the related works discuss only stack allocating objects whereas our approach also considers garbage collection improvement basing on the compile-time analysis.

6 Conclusion

This paper presented a technique for fast and scalable object lifetime analysis. Being used in cooperative compiler and run-time framework, the implemented optimizations profit in both execution speed and memory consumption of Java applications. The interesting area for future works is to investigate a region inference algorithms allowing compiler to approximate object lifetimes between method call boundaries. Despite the applicability of such analysis to compiler optimizations is doubt, the information may be used for more effective garbage collection in compiler-cooperative run-time environment.

References

1. J. Gosling, B. Joy and G.Steele: *The Java Language Specification.* Addison-Wesley, Reading, 1996
2. Gu et al.: *The Evolution of a High-Performing JVM.* IBM Systems Journal, Vol. 39, No. 1, 2000
3. *The Java HotSpot(tm) Virtual Machine*, Technical Whitepaper, Sun Microsystems Inc., Whitepaper, 2001.
http://www.sun.com/solaris/java/wp-hotspot
4. D. Detlefs, T. Printezis: *A Generational Mostly-Concurrent Garbage Collector.* In Proc. ACM ISMM'00, 2000
5. V.V. Mikheev: *Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution* In Proc. Joint Modular Languages Conference, JMLC'2000, Volume 1897 of LNCS, Springer-Verlag, 2000
6. J.T. Shwartz: *Optimization of very high-level languages -I. Value transmission and its colraries.* Computer Languages, 1(2), 1975
7. D. Gay, B. Steensgaard: *Stack Allocating Objects in Java.* Reasearch Report, Microsoft Research, 1999
8. J.-D. Choi et al.: *Escape Analysis for Java.* SIGPLAN Notices, Volume 34, Number 10, 1999
9. B. Blanchet: *Escape Analysis for Object Oriented Languages. Application to Java(tm).* SIGPLAN Notices, Volume 34, Number 10, 1999
10. D. Chase, M. Wegman, and F. Zadeck: *Analysis of pointers and structures.* SIGPLAN Notices, Volume 25, Number 6, 1990
11. D. Bacon: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* Technical Report, University of California, 1997

12. D. Bacon, P. Sweeney: *Fast static analysis of C++ virtual calls.* In Proc. OOP-SLA'96, SIGPLAN Notices, Volume 31, Number 10, 1996
13. O. Agesen: *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism.* In Proc. ECOOP'95, Aarhus, Denmark, 1995
14. A. Deutch: *On the complexity of escape analysis.* In Conference Record of POPL'97, The 24th ACM SIGPLAN-SIGAST, 1997
15. S. Hughes: *Compile-time garbage collection for higher-order functional languages.* Journal of Logic and Computation, 2(4), 1992
16. K. Inoue et al.: *Analysis of functional programs to detect run-time garbage cells.* ACM Transactions on Programming Languages and Systems, 10(4), 1988
17. Y. Park, B. Goldberg: *Escape analysis on lists* In Proceedings of PLDI'92, ACM SIGPLAN, 1992
18. *JET Deployment Environment.* Excelsior LLC, Technical Whitepaper, 2001. **http://www.excelsior-usa.com/jetwp.html**

A Software Composition Language and Its Implementation

Dietrich Birngruber

Johannes Kepler University Linz,
Institute for Practical Computer Science
System Software Group
A-4040, Linz, Austria
birngruber@ssw.uni-linz.ac.at

Abstract. The actually achieved benefits of using binary software components are not as revolutionary as promised. Component platforms are available but the composition process is not "componentized". We propose to increase the automation of software composition along with the necessary degree of flexibilty by introducing a set of languages (CoPL and CoML) and tools. By automating the composition process routine tasks are performed by tools. Additionally, software engineers can preserve and instantiate composition patterns in CoPL and CoML and thus at a higher level of abstraction than at the level of pure glue code.

1 Introduction

Binary software components offer solutions to various software engineering problems, e.g. how to build and maintain complex software systems in a changing environment. The idea is to acquire prefabricated, well-tested and platform independent binary software components on the market and to compose them to new applications by plugging them together in builder tools without the need for coding. There are already markets [1] for components as well as some common understanding about the term software component [2]:

> A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

The composition of binary software components divides the development process into two parts. First, component developers write new component libraries and they distribute or *deploy* the components over a market. Second, *third parties* like e.g. application programmers use them to compose their applications. Often different individuals assume these roles. This leads to a knowledge gap, as the application programmer has to determine how and in which context he can apply the different components. Of course, a component provider has to state what the components need in order to work. The component provider defines the component's *context dependencies* in a (hopefully) proper documentation and often he additionally provides example code, too.

With Component Plan Language (CoPL) and Component Markup Language (CoML) we try to provide tools for the last part of the above component definition: tools for component *composition by third parties*.

Despite the used component definition we had to investigate the technical requirements of a component — being more precise: which requirements components need to be composable with CoPL and CoML:

- A binary component is accessed via public interfaces.
- A component provides one or more strongly typed interfaces.
- A single interface offers methods and / or properties. Properties can be read and set.
- The primary composition model is connection oriented and this model is implemented by an event mechanism.
- The component life-cycle is split into design-time and run-time and thus between wiring components versus creating instances.

The requirements are oriented on industry component technology standards like Microsofts COM [7] and .NET [9], Suns JavaBeans [4] and Enterpirse JavaBeans [5], and OMGs CORBA [6].

Before we present CoPL and CoML, we look at the motivation behind our approach. Why we did not use existing approaches is explained in section 5. Finally, in section 6 we draw some conclusions and outline the plans for the future.

2 Motivation and Usage Scenario for CoPL and CoML

In this section we present the motivation or the basic ideas behind our approach. We introduce the ideas of component plans and Decision Spots for improving the automation of the component composition process.

A component plan describes how an application programmer typically glues components of a given library together. A plan is a description of a composition with optional Decision Spots. Typically a plan is written in the Component Plan Language (CoPL) and captures domain knowledge and typical usage scenarios or composition patterns by providing a typical pre-wiring of the used components. The application programmer processes the CoPL plans with a CoML generator. The generator produces a detailed component composition description in Component Markup Language (CoML) code. CoML is an XML [3] application which is component platform and tool independent. Figure 1 shows a typical usage scenario for CoPL and CoML.

The generator uses the plan as input and — if stated in the plan — asks the application programmer to substitute the declared place-holders by concrete components from a list of matching component implementations. We call these place-holders Decision Spots. Currently the matching algorithm is based on type substitutability or on syntax and not on semantic information.

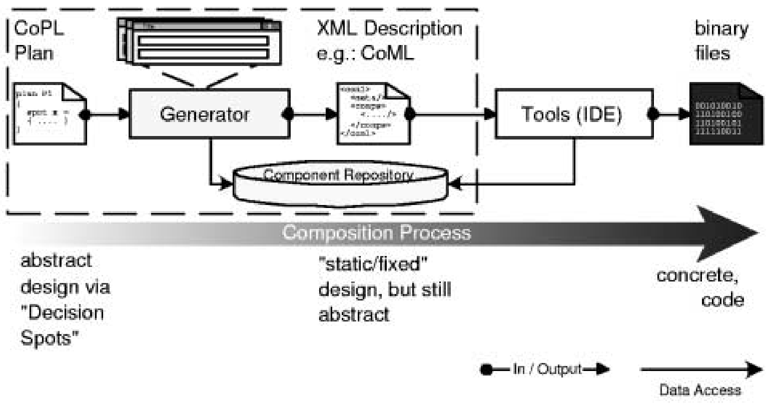


Fig. 1. CoPL and CoML Usage Scenario

On the one hand writing glue code manually gives the application assembler great flexibility, where on the other hand tools (e.g. wizards) automate routine and clearly predefined composition tasks, like generating a code snippet for a new GUI dialog. A possible way for combining the advantages of these composition techniques is to introduce a "script-able generator". In fact, a CoPL plan is used for scripting the CoML generator. The interpreted plan guides application programmers semi-automatically (similarly to a wizard) through the assembly process for example by displaying a dialog for choosing the desired implementation (e.g. ArrayListImpl) for a given interface (e.g. for an "IList" interface). In contrast to a wizard, a plan is not fixed but can be modified. It is like a composition template with some degrees of freedom. A plan may contain Decision Spots that offer choices to the application programmer.

Considering a library with many components, it is a tedious task to find the right components and to instantiate and glue them together according to the desired composition pattern. Our component plans along with the generator automate this process by supplying the programmer with knowledge about how the component developer intended to wire the components.

CoPL is based on previous work on JavaBeans composition using plans (see [8]). However, CoPL and CoML are not tuned toward a special component technology like JavaBeans, or Microsoft's .NET components.

3 CoPL — Component Plan Language

Component plans describe component compositions at an abstract level. A plan describes which components are used, how they are configured and how they are composed. A plan can contain Decision Spots. CoPL code is used for controlling the output of the CoML generator. CoPL is executed at design time and not at run time like other composition and scripting languages (e.g. like JavaScript [13], Piccola [10]).

The predecessor of CoPL was the JavaBeans oriented bean plan. The syntax of a bean plan is oriented on the Syntax of C++, C# or Java because we consider the syntax of widely used general purpose programming languages more readable than XML. However, currently we are migrating the bean plan language to the platform independent Component Plan Language and we are considering to use XML as CoPLs meta schema.

Example 1. The following simple example uses the C like syntax. In this plan we are combining a collection component with a printer component. MyCollections.IList represents a list interface. Whenever a component is added to the list, the Change event is fired. The component "out" handles this event.

```
plan ListWithPrinterExample {
  //simple decision spot
  spot ListImpl = <MyCollections.IList>;
  comp out = new Test.Printer {}
  comp list = new ListImpl {
    Size = 5; //set the property
    on Change call out.PrintLn("event␣fired␣:-)");
  }
}
```

CoPL allows the description of components, properties, methods, events, aggregation and of Decision Spots. Table 1 gives an overview of the offered abstractions in CoPL.

The Decision Spots give the plans the needed flexibility and are explained in more detail. Example 1 shows the use of a simple Decision Spot (ListImpl). The spot declares the used interface IList and not the exact implementation. It offers the application programmer the choice to select one concrete MyCollections.IList implementation. The creator of a plan uses interactive Decision Spots whenever he does not know the exact type or implementation of a component. He delegates the final decision to the application programmer who runs the CoML generator. The result of a Decision Spot is either the name of a concrete component implementation or null. If the result of the spot is used to create a new component instance (like a new list) and the spots value is null, the instance is not used.

A Decision Spot rule declares one or several choices. A choice declares an interface and the CoML generator has to find those components that implement the given interfaces. Interfaces are organized in an object oriented manner: They are part of an inheritance hierarchy. Thus, the rule of a decision spot can narrow the search vertical and horizontal (see Figure 2). A vertically search is limited to a particular inheritance branch. An example is the search for all implementations of the IList interface (B) which also includes the implementations for the IDictionary interface (C). A horizontally search can cover all implementations of a given set of interfaces. They don't need to belong to the same base type. An

Table 1. CoPL overview

Abstraction	Description
Plan	The plan itself. A plan contains several declared components and Decision Spots. The components are connected via events or via aggregation.
Component	The description of the used component implementation and interface.
Event	Associates event source (component one), event and event handler (compo-nent two) with each other.
Containment	Aggregation based composition description for adding components to a container.
Decision Spot	A spot consists of a rule for finding a concrete component implementation, or for selecting the value for a certain property of a component.
Method and property	For setting and getting the properties of a used component and calling arbitrary methods.
Customizer and property editor	Some component models (JavaBeans, COMs ActiveX controls and successors) introduce special editors for customizing a component interactively during design time.

Example is the search for implementations of ICollection (A) and for ButtonCtrl types (D).

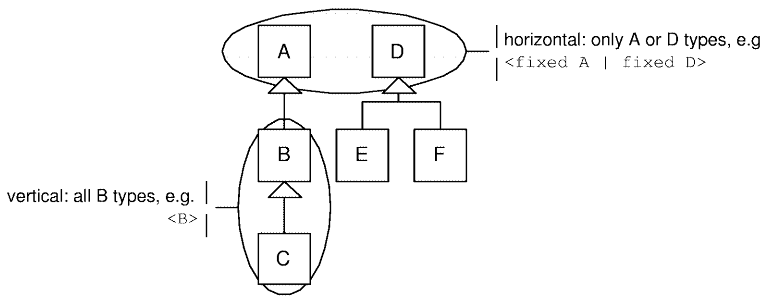

A
D
horizontal: only A or D types, e.g
<fixed A | fixed D>
B
E
F
C
vertical: all B types, e.g.
<B>

Fig. 2. Horizontal and vertical search within the type hierarchy

Decision Spots can depend on each other. The result of a Decision Spot can influence the set of choices of another Decision Spot. We defined a simple dependency check (like an if statement) where the set of choices of a spot s2 can vary depending on the selection of a previous executed spot s1.

Example 2. This example shows the horizontal and vertical selection mechanism. The search rule covers three types and is encoded within angle brackets (`< >`). The horizontal search alternatives are separated by a —.

```
spot layout = <FlowLayout | BorderLayout |
               fixed CardLayout>;
```

The Decision Spot in example 2 gives the application programmer the choice between three possible components given by the types FlowLayout, BorderLayout, and CardLayout. When the CoML generator encounters this spot, it opens a dialog containing all implementations of the three types with all available subtype implementations of the first two alternatives. The application programmer can select on of the implementations. If the implementation of possible subtypes should be omitted like for CardLayout, one has to insert the keyword `fixed`. Only those implementations are offered as a choice which directly implement CardLayout.

Decision spots allow the dynamic extensibility of plans. Subtypes and their implementations, like e.g. MyBorderLayout, known only to the application programmer are displayed in the selection dialog, even though the creator of the plan does not know them.

4 CoML — Component Markup Language

The Component Markup Language (CoML) is an XML application for composing software components. The main goal for CoML is to have a platform independent description of a component composition which is process-able by various software tools like development tools. CoML can be interpreted like other scripting languages. In our intention CoML should primarily be created and used by software tools rather than require humans to manually write (and execute) CoML scripts. However, in the spirit of XML, we still tried to make CoML human readable as well and developed an interpreter for the Java and the .NET platform.

In section 1 we introduced the term component. We stated requirements for components and for composition languages. Based on this requirements, CoML offers abstractions (i.e. XML tags, see also Table 2) for describing a component composition. CoML tags can be used for:

– describing which component library and version are used and additional component platform dependent information (see `meta` and `info` elements);
– describing a component itself by stating the desired interface and implementation (see `components` and `component` elements);
– describing the customization of a component (see `property` and `method-call` elements);
– describing the component composition (see `on-event` and `add` elements).

Example 3. This CoML snippet shows the composition of two components: a list component and a printer component (see also the CoPL example 1). The list component fires a change event, whenever an item is added to the list. The printer component reacts on the change event and prints "event fired :-)" on the console.

Table 2. CoML elements (tags)

Element name	Description
`coml`	root element
`meta`	contains the meta information
`components`	list of composed components; contains ¡component ...¿ elements
`component`	declares a component
`property`	sets or gets a property of a specific component
`method-call`	calls a method
`on-event`	describes an event binding; connection based composition
`add`	adds another component; aggregation based composition
`string`, `int`, ...	elements for primitive data types

Additionally, each element has several attributes.

```
<?xml version="1.0" encoding="utf-8" ?>
<coml name="ListWithPrinterExample" version="0:0:0:1">
  <meta>
    <info type="java" version="1.3">
      <!-- additional (linker) info:  -->
      <!-- used libraries + versions -->
    </info>
  </meta>
  <components>
    <component id="out" class="Test.Printer" />
    <component id="list" interface="MyCollections.IList"
     class="Test.ListWithChangeEvent">
      <property name="Size" access="set">
        <int>5</int>
      </property>
      <on-event name="Change">
        <method-call idRef="out" name="PrintLn">
          <string>event fired :-)</string>
        </method-call>
      </on-event>
    </component>
  </components>
</coml>
```

Example 3 shows the structure of each CoML script: The root element `coml` contains exactly two children — `meta` and `components`. The `components` element describes the actual component composition and the `meta` element provides additional platform specific information and the used version. The elements abstract from the target platform. It depends on the tool which processes CoML if the `meta` element is used or not. An interpreter may care about the used component libraries and it may define version compatiblity rules.

The `meta` element describes platform specific informaiton and the compatible platform version. It consists of a list of `info` elements. An `info` element describes which component library and version are used. It can contain elements which describe platform specific information. By separating the actual component composition from meta information and platform specific information CoML is extensible and can be adjusted to other usage scenarios and tools.

The `components` element describes the actual software composition. It consists of a list of `component` elements. The `component` element declares a single component. It declares the component class, the used interface and the customization. The child elements of the `component` element describe the customization. Proper child elements are the `add`, the `property`, the `method-call` and the `on-event` element. However, the `component` element itself can be a child element of the `add`, `property` or `method-call` element. If the `component` element is used as a child element, then it describes the value or actual parameter of the enclosing element (like the parameter of a method call).

Generally, the child elements are the arguments of the enclosing parent element. The syntax of some CoML elements is influenced by IBMs Bean Markup Language.

5 Related Work

Sun and IBM have developed their own composition language based on XML. However both, Sun's JavaBean Persistence [11] and IBM's Bean Markup Language (BML), are tailored for JavaBeans and do not support meta information at all.

The main goal of Sun's approach is to have a proprietary standardized format for exchanging mainly GUI JavaBeans compositions between different Java IDEs. An idea similar to CoMLs primary purpose. At the beginning of the project we tried to use Bean Persistence. Unfortunately Bean Persistence expressiveness for composing components via events is too limited. Bean Persistence uses special proxy classes to build an event connection and does not provide XML elements to clearly mark an event connection. Bean Persistence depends on the JavaBean mechanism for detecting event sources — i.e. method naming conventions — and does not abstract the actual event-gluing. The original JavaBeans specification does not define a logical containment/hierarchy relationship among connected beans and Bean Persistence does not support it either.

CoML is influenced by BML. The main differences are that CoML is not focused on JavaBean composition, that CoML supports interfaces where BML allows explicit type conversions, that CoML does not allow embedding of foreign scripting code — such as JavaScript — in order to remain platform independent, and that CoML supports meta information.

During the last years several software composition approaches emerged. Approaches like Generative Programming [14] which subsume besides other methods Aspect Oriented Programming [15], Generic Programming [16], and Software Product Lines [17].

C++ and Ada programmers know what generics or templates (like the STL [18]) are. Some of the main research interests of generic programming are how to design generic components and algorithms with the appropriate theory and formalism behind it, and which software technologies (like C++ templates) for programming generic components are usable. Very generic components are hard to customize and very specific components are hardly reusable, because they are tailored for one special purpose.

One characteristic of the evolution to the industrial age is mass production. Henry Ford introduced the product line when he manufactured the Ford T Model for the mass. Software Product Lines try to use this paradigm for producing a family of software. The members of the same family have a lot of common features and only some variant functionality like the variations in a control software for a certain family of brick and mortar diesel engines. Software product lines cover the whole software engineering process and the necessary organizational changes for creating a family of software and their variants. They consider components as reuse in the small and product lines as reuse in the large. Some of the main research interests are what parts (or functionality) of a components interface(!) are immutable and what are variants. An academic example for a product line architecture is the GenVoca architecture [19].

Formal specifications and the derivative composition approach are research topics which evolved from the AI and knowledge based software engineering fields. They are subsumed under Automated Software Engineering [20]. Their ambitious goal is to derivate and generate the application from the specification.

Our approach (CoPL and CoML) does not try to change the available components by splitting them up in a core part and a variable part, like the product line approach does. It tries to automate the selection of possible implementations. It shares the idea of automatically composition of already existing components. Our approach can be seen as a contribution to the approaches that share this vision, too.

6 Conclusions

An application programmer uses component plans at design-time, i.e. when he assemblies the components to a new application. The benefits are to have a script-able wizard, that produces a platform independent description of a concrete component composition.

Plans along with the CoML generator are used at a different point in time than scripting languages, which are typically interpreted or executed (like e.g. Piccola, JavaScript or IBM's Bean Markup Language). These languages are interpreted at run-time, i.e. at the end user's computer during actual execution of the application.

The output of the generator is a composition description in CoML. CoML is component technology and platform independent. Different tools like development tools, documentation tools or software architecture visualizing tools can use CoML, e.g., for exchanging component compositions or displaying them in different manners. Of course, CoML can be interpreted as well and currently we have interpreters for Java and Microsoft's .NET component platform.

CoPL and CoML are an ongoing research project and not finished yet. Some of future research efforts for CoML cover how to handle different formal parameter behavior of method calls and how to provide a convenient error handling mechanisms. It is still not resolved which information should be part of the meta information and whether we will address cross platform composition as well. We consider to use an XML based syntax for CoPL and if the selection mechanism of the Decision Spot should cover semantic criteria as well. We also need to improve our tools (CoML generator etc.) and to integrate them in existing development environments.

References

1. `http://www.developer.com/directories`
2. Szyperski Clemens: Component Software — Beyond Object-Oriented Programming. Addison-Wesley. 1997.
3. W3C: Extensible Markup Language (XML) 1.0. W3C Recommendation. 10. Feb. 1998. `http://www.w3.org/TR/REC-xml`.
4. Sun Microsystems: JavaBeans API Specification. 1997. `http://java.sun.com/beans/spec.html`.
5. Sun Microsystems: Enterprise JavaBeans(TM) 2.0 Specification. Proposed Final Draft 2. 2001. `http://java.sun.com/products/ejb/`.
6. Object Management Group: The Common Object Request Broker: Architecture and Specification. Editorial Revision: CORBA 2.4.2. February 2001. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`.
7. Microsoft: The Component Object Model Specification. 1995. `http://www.microsoft.com/com/resources/comdocs.asp`
8. Birngruber Dietrich, Hof Markus: Using Plans for Specifying Preconfigured Bean Sets. In: Li Qiaoyun, et. al. (Eds.): Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34). Santa Barbara, CA. 2000.
9. Richter Jeffrey: .NET Framework: Building, Packaging, Deploying, and Administering Applications and Types. in: MSDN magazine. February 2001.
10. Achermann Franz, Nierstrasz Oscar: Applications = Components + Scripts – A tour of Piccola. in: Mehmet Aksit (Ed.): Software Architectures and Component Technology. Kluwer. 2000.
11. Milne Philip, Walrath Kathy: Long-Term Persistence for JavaBeans. Sun Microsystems. 24. Nov. 1999.
12. Curbera Francisco, Weerawarana Sanjiva, Duftler Matthew J.: On Component Composition Languages. in: Bosch, Szyperski, Weck (Ed.): Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000). 2000. ISSN 1103-1581. see also `http://www.alphaWorks.ibm.com/formula/bml`.
13. ECMA 262: ECMAScript Language Specification. 3rd Edition. December 1999. `http://www.ecma.ch`.
14. Czarnecki Krzysztof, Eisenecker Ulrich W.: Generative Programming. Methods, Tools, and Applications. Addison Wesley. 2000.
15. Kiczales Gregor, Lamping John , Mendhekar Anurag, Maeda Chris, Lopes Cristina V., Loingtier Jean-Marc, Irwin John: Aspect-Oriented Programming. in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Finland. LNCS 1241. Springer. June 1997.

16. Jazayeri Mehdi, Loos Rüdiger G.K., Musser David R. (Eds.): Generic Programming. International Seminar on Generic Programming. Selected Papers. Dagstuhl Castle Germany. LNCS 1766. Springer. April/May 1998.
17. Donohoe Patrick (Ed.): Software Product Lines. Experience and Research Directions. Proceedings of the First Software Product Line Conference (SPLC1). Denver USA. Kluwer. August 2000.
18. Austern Mathew: Generic Programming and the STL. Addison Wesley. 1999.
19. Batory Don: Subjectivity and GenVoca Generators. in: Proceedings of the 4th International Conference on Software Reuse (ICSR 96). Orlando USA. IEEE. April 1996.
20. Alexander Perry, Flener Pierre (Eds.): Proceedings of ASE-2000: The 15th IEEE Conference on Automated Software Engineering. Grenoble France. IEEE CS Press. September 2000.

Editor Definition Language and Its Implementation

Audris Kalnins, Karlis Podnieks, Andris Zarins, Edgars Celms, and Janis Barzdins

Institute of Mathematics and Computer Science,
University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{audris, podnieks, azarins, edgarsc, jbarzdin}@mii.lu.lv

Abstract. Universal graphical editor definition language based on logical metamodel extended by presentation classes is proposed. Implementation principles of this language, based on Graphical Diagramming Engine are described.

1 Introduction

Universal programming languages currently have become more or less stable, the main effort in this area is oriented towards improving programming environments and programming methodology. However, the development of specialised programming languages for specific areas is still going on (most frequently, this type of languages is no more called programming languages, but specification or definition languages). One of such specific areas is the definition of graphical editors. The need for various graphical editors and similar tools based on graphical representation of data increases all the time, because due to increased computer speeds and size of monitors it is possible to build graphical representations for wider subject areas. In this paper the **Editor Definition Language (EdDL)** for a simple and convenient definition of wide spectrum of graphical editors is proposed, and the basic implementation principles of EdDL are described.

Let us mention some earlier research in this area. Perhaps, the first similar approach has been by Metaedit [1], but its editor definition facilities are fairly limited. The most flexible definition facilities (and some time ago, also the most popular in practice) seem to be the Toolbuilder by Lincoln Software. Being a typical meta-CASE of early nineties, the approach is based on ER model for describing the repository contents and special extensions to ER notation for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. The diagram itself is being defined by means of a frame describing graphical objects in it and the supported operations. A more academic approach is that proposed by Kogge [2], with a very flexible, but very complicated and mainly procedural editor definition language. Another similar approaches are proposed by DOME [8] and Moses [9] projects, with fairly limited definition languages. Several commercial modelling tools (STP by Aonix, ARIS by IDS prof. Scheer etc) use a similar approach internally, for easy customisation of their products, but their definition languages have never been made explicit.

Our approach in a sense is a further development of the above-mentioned approaches. We develop the customisation language into a relatively independent editor definition language (EdDL), which, on the other hand, is sufficiently rich and easy to use, and, on the other hand, is sufficiently easy to understand. At the same time it can be implemented efficiently, by means of the universal Editor Engine. Partly the described approach has been developed within the EU ESPRIT project ADDE [3], see [4] for a preliminary report.

2 Editor Definition Language. Basic Ideas

The proposed editor definition language consists of two parts:

– the language for defining the logical structure of objects which are to be represented graphically
– the language for defining the concrete graphical representation of the selected logical structure.

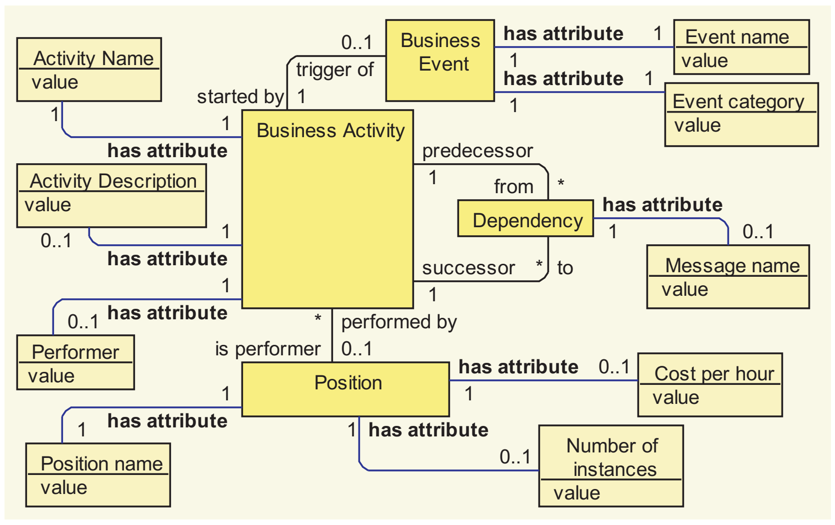


Fig. 1. Logical metamodel example

The separation of these two parts is the basis of our approach. For describing the logical structure there exists a generally adapted notation — by UML class diagrams [5], which typically is called the **logical metamodel** in this context. Fig.1 shows a simple example of a logical metamodel for business activities. *Business activities* are assumed to be parts of a larger process. There may be a *dependency* between business activities (this dependency may be a *message* passed from one activity to another, but also something more general). An activity may be triggered by an (external) *business event*. Business activity may have a *performer* — a *position* within a company. This example will be used in the paper to demonstrate the editor definition features. Fig.1 needs one technical remark to be given. Attributes of a class there are extracted as separate classes, linked via an association with the predefined role name **has attribute** to the parent class. This attribute extraction is technically convenient for defining the presentation language part. Otherwise the logical metamodel is an arbitrary UML class diagram.

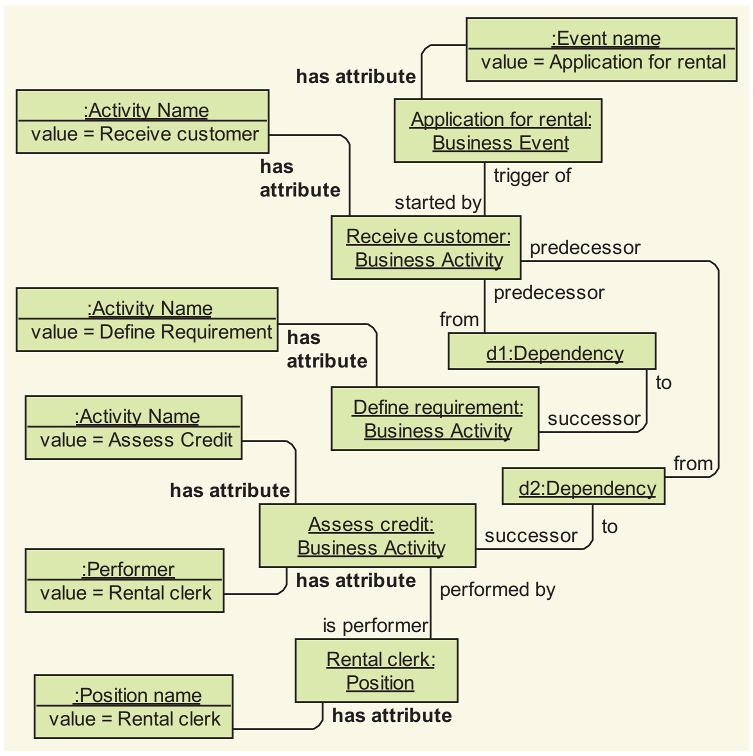


Fig. 2. Example of instance diagram

Fig.2 shows an example of an instance diagram (object diagram in UML terms) corresponding to the logical metamodel in Fig.1. Our goal is to define the corresponding editor, which in this case could be able to present the instance diagram as a highly readable graphical diagram in Fig.3 (where the traditional rendering of dependencies by oriented lines is used). A special remark should be given with respect to *Position*. It is not explicitly represented in diagram in

Fig. 3, but double- clicking on the performer name is assumed to **navigate to** (i.e. to open) a special editor showing the relevant position (this other editor is not specified here). The **navigation** and the **prompting** (the related action by means of which such a reference can be easily defined) are integral parts of our editor definition facilities.

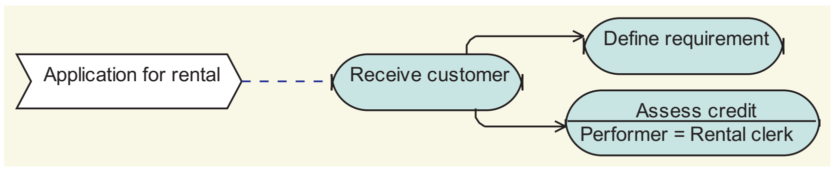


Fig. 3. Business activity diagram example

Roughly speaking, the goal of our definition language is to describe the translation of pictures like Fig. 2 into equivalent pictures like Fig. 3. So it is a sort of graphics translation language.

Now let us start a detailed outline of the EdDL. Like any real language, it contains a lot of details, therefore we will concentrate only on the basic constructs. The language will be presented as an extension of the logical metamodel adhering to the UML class diagram notation. Fig.4 demonstrates the use of EdDL for the definition of the example editor (with some minor details omitted). In this figure rectangles represent the same classes from the logical metamodel in Fig. 1, but rounded rectangles represent classes being the proper elements of EdDL. Classes with class names in bold represent abstract classes, which cannot be modified (they are used mainly for inheritance). Similarly, bold role names of associations represent the fixed ones. We remind that the underlined attributes (to be seen in EdDL classes) are class attributes (the same for all instances) according to UML notation.

The first element added to the logical metamodel is the **diagram** class (*Business activity diagram*), together with standard associations (with the role name **contains**) to the contained diagram objects, and as a subclass of the fixed *Diagram* class. One more standard association for diagram is the refinement association (**refines**), which defines that a *Business Activity* can be further refined by its own *Business activity diagram* (this definition is sufficient for the Editor Engine to enable this service).

Each of the metamodel classes, which must appear as graphical objects in the diagram, are linked by an unnamed association to its **presentation class** — a subclass of standard classes **box** or **line**. The presentation class may contain several class attributes (with predefined names). Thus the presentation class for *Business Activity* — the *Activity box* class says that every business activity must be represented by a rounded rectangle in a light blue default colour. The *Icon* representing this graphical symbol on the editor's **symbol palette** (to create a new business activity in a diagram) is also shown. For presentation classes being lines the situation is similar, but there may be lines corresponding to associations in the metamodel (*Triggering line*) or to classes (*Dependency line*). The latter case is mostly used for lines having associated texts in the diagram (corresponding to attributes of the class; here the *Message name*). For showing the direction of line (and other similar features) the relevant role names from the metamodel are referenced in the presentation class (e.g. **start**=*predecessor*).

Class attributes are being made visible in diagrams by means of a **compartment** presentation class. The most important attribute of a compartment class is the **position** — for boxes in the simplest case it means the number of the horizontal slice of the box, for lines it means the relative positioning (start, middle, end). Compartment class may contain also style attributes (visible **tag**, separator, font, etc.).

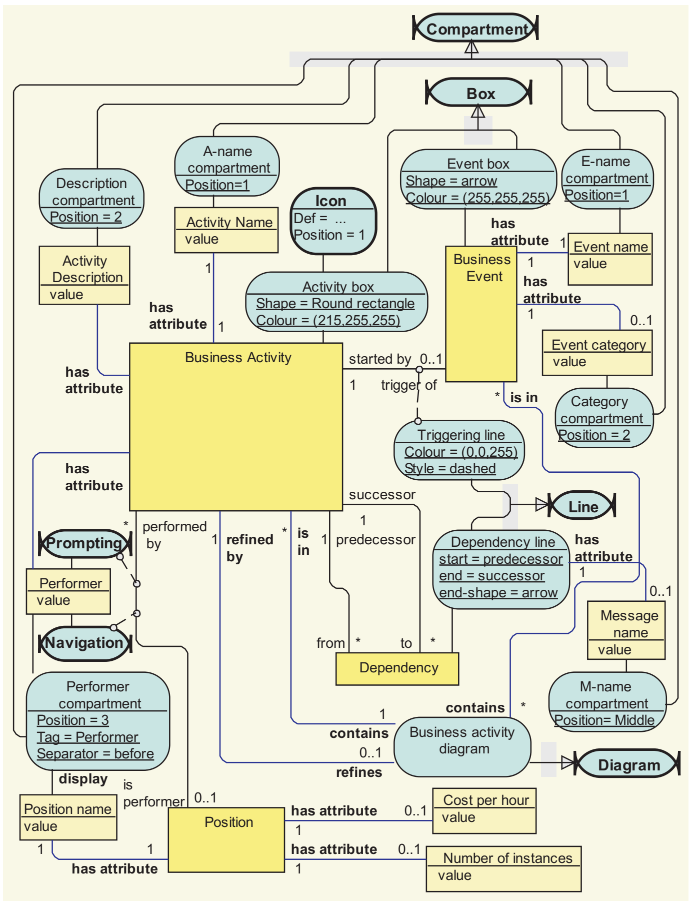

Compartment
Box
Line
Diagram
Prompting
Navigation
Icon
Def = ...
Position = 1
Description compartment Position = 2
Activity Description value
A-name compartment Position=1
Activity Name value
Activity box Shape = Round rectangle Colour = (215,255,255)
Event box Shape = arrow Colour = (255,255,255)
E-name compartment Position=1
Event name value
Business Event
Event category value
Category compartment Position = 2
Business Activity
Triggering line Colour = (0,0,255) Style = dashed
Dependency line start = predecessor end = successor end-shape = arrow
Message name value
M-name compartment Position= Middle
Dependency
Business activity diagram
Performer value
Performer compartment Position = 3 Tag = Performer Separator = before
Position name value
Position
Cost per hour value
Number of instances value
has attribute
started by
triggers of
is in
successor
predecessor
performed by
refined by
is in
from
to
contains
refines
display
is performer
0..1
1
*

Fig. 4. Business activity diagram editor definition in EdDL

The most interesting element in this EdDL example is the definition of **prompting** and **navigation**. They are both related to the situation when an attribute (i.e. its value) of a metamodel class (the *Performer* for *Business activity* in the example) actually is determined by an association of this class (a derived attribute in UML terms). Here the *Performer* value actually must be equal to the *Position name* of that *Position* instance (if any) which is linked by the association having the role name *Performed by* (the fact that a *Position* must be represented by its *Position name* is defined by the display association). Prompting here means the traditional service found in an editor that a value can be selected from the offered list of values (value of *Performer* selected from the list of available *Position names*), with the automatic generation of the relevant association instance as a side effect. The navigation means the editor feature that double-clicking on the *Performer* field (which presents the *Performer* attribute) in a *Business activity* box automatically invokes some default editor for the *Position* (the target of the association). Both *Prompting* and *Navigation* are shown in the EdDL as fixed classes linking the attribute to the relevant association (they may have also their own attributes specifying, e.g. the prompting constraints). Note that *Position* has no presentation class in the example, consequently its instances are not explicitly visible in the *Business activity diagram.*

Certainly EdDL contains more features than demonstrated in Fig.4, e.g. various uniqueness constraints, definitions for attribute "denormalisation", modes of model/diagram consolidation etc. The EdDL coding shown in Fig. 4 was simplified to make it more readable. The actual coding used for the commercial version of EdDL is much more compact, here the metamodel class attributes are defined in the traditional way, and most of Presentation classes are coded just as UML constraints (properties) inside a metamodel class. Nevertheless the semantics of this language is just the one briefly described in the paper. We assert that EdDL is expressive enough to define practically any types of editor that could be used to build related sets of diagrams in system modelling area. Namely the inter-diagram relations such as prompting and navigation are the most complicated ones, and they successfully managed in EdDL. Finally, Fig. 5 shows the defined editor in action.

3 EdDL Implementation Principles

EdDL has been implemented by means of an interpreter which in this case is named **Editor Engine**. When an editor has been defined in EdDL the Editor Engine acts as the desired graphical editor for the end user. Here only the main principles of implementation will be discussed. The first issue is the information storage. It is universally accepted nowadays that the logical metamodel describes directly (or very close to it) the physical structure of the tool repository. This repository can be an OODB, a special tool repository (e.g. Enabler [6]) or a relational DB. This is one more argument why the editor definition should be based on a separately defined metamodel.

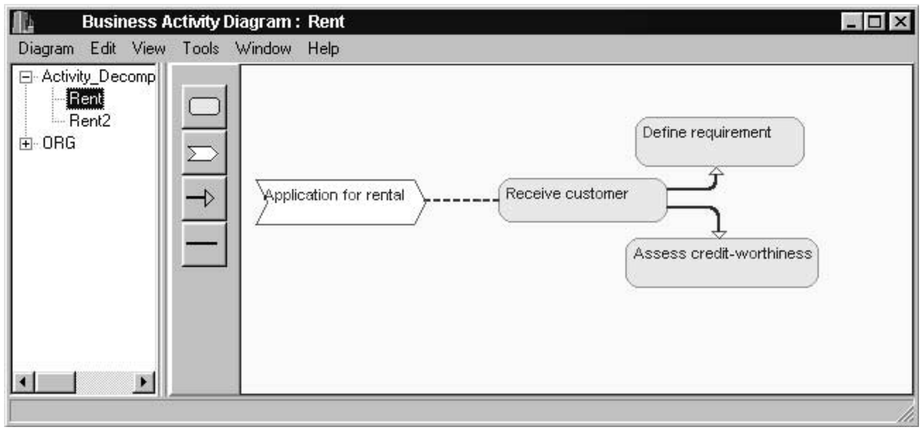


Fig. 5. Editor in action

Fig. 6 shows the general architecture of the EdDL approach. A key aspect is that Editor Engine (EE) relies on **Graphical Diagramming Engine (GDE)** for all diagram drawing related activities. The primitives implemented by GDE — diagram, box, line, compartment etc. and the supported operations on them are very fit for this framework.

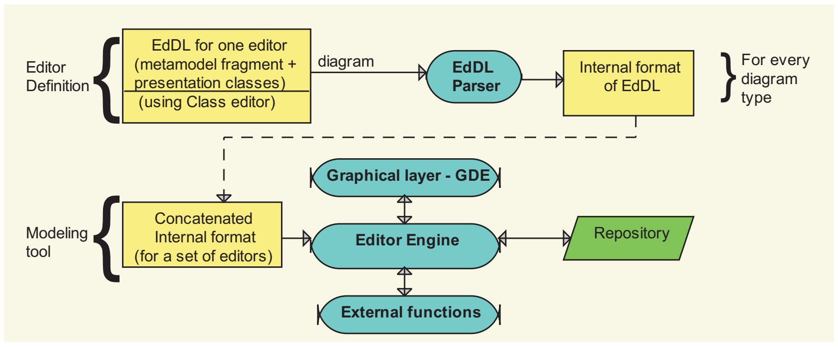


Fig. 6. Architecture of EdDL implementation

Thus the interface between EE and GDE is based on very appropriate high level building blocks, there is no need for low level graphical operations in EE at all. The GDE itself was developed by IMCS UL initially within the framework of ADDE project, with a commercial version later on. It is based on very sophisticated graph drawing algorithms [7].

The general technology of using EdDL for defining a set of editors is quite simple. For each of the editors its definition on the basis of the relevant metamodel fragment is built. Then these definitions are parsed and assembled into one set. EE "performs" this set, behaving as a modelling tool containing the defined set of diagrams. A lot of tool functionality — "small operations" such as copy-paste and "large operations" such as load and save are implemented in EE in a standard way and need no special definitions. Certainly, external modules can be included for some specific operations.

The practical experiments on using EdDL and EE have confirmed the efficiency and flexibility of approach. The defined editors (like the one in Fig.5) behave as industrial quality graphical editors. The flexibility has been tested by implementing full UML 1.3 and various specific business process related extensions to it.

4 Conclusions

The paper presents a brief outline of the graphical editor definition language EdDL and its implementation. But we can view all this also from a different angle. Actually a new kind of metamodel concept application for a specific area — editor definition — has been proposed. However this approach can be significantly more universal, since it is generally accepted that object model (Class diagram) is a universal means for describing the logical structure of nearly any system. Thus the same approach of extending this model by special "presentation" classes could be used, e.g. to define model dynamics, simulation etc., but this is out of scope for this paper.

References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit — a flexible graphical environment for methodology modelling. Springer-Verlag (1991)
2. Ebert, J., Suttenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain (1997)
3. ESPRIT project ADDE. http://www.fast.de/ADDE
4. Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: Towards a Metamodel-Based Universal Graphical Editor. Proceedings of the Third International Baltic Workshop DB&IS, Vol. 1. University of Latvia, Riga, (1998) 187-197
5. Rumbaugh, J., Booch, G., Jackobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)
6. Enabler Concepts Release 3.0, Softlab GmbH, Munich (1998)
7. Kikusts, P., Rucevskis, P.: Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors. Lecture Notes in Computer science, Vol. 1027. Springer- Verlag (1996) 361-364
8. DOME Users Guide. http://www.htc.honeywell.com/dome/support.htm
9. Esser, R.: The Moses Project.
http://www.tik.ee.ethz.ch/ moses/MiniConference2000/pdf/Overview.PDF

Oberon-2 as Successor of Modula-2 in Simulation

Alexey S. Rodionov[1] and Dmitry V. Leskov[2]

[1] Institute of Computational Mathematics and Mathematical Geophysics
Russian Academy of Sciences, Siberian Branch
phone: +383-2-396211, alrod@rav.sscc.ru
[2] A. P. Ershov Institute of Informatics Systems,
Russian Academy of Sciences, Siberian Branch
6, Acad. Lavrentjev ave., 630090, Novosibirsk, Russia

Abstract. In the report the programming languages Oberon-2 is discussed from the point of view of convenience to program the discrete event simulation systems (DESS). Its predecessor Modula-2 was used as the basic language for a number of simulation packages and was proved to be good for it, but has Oberon-2 enough features to replace it and stand against domination of C++ in this special area? Are there compilers and programming environments good enough for this purpose? Is it possible to use existent libraries and transfer software between different platforms? These and other questions are discussed on the examples of ObSim-2 simulation package and XDS Modula-2&Oberon-2 programming system.

1 Introduction

The programming language Modula-2 for a long time has attracted attention of the developers of DESS. It was proved to be a convenient tool for the development of well-structured programs with the possibility of organization of quasi-parallel processes. The later is of a special importance for Simula-like DESS. A number of simulation packages on Modula-2 were designed [1]-[7]. Moreover, Modula-2 was proved to be so good programming language for simulation packages that it was used as a basis for the design of special simulation languages. For example, one the most powerful simulation systems of the late, MODSIM III [12], has Modula-like language.

One of the authors designed the package SIDM-2 [8] using Modula-2 as the basic language.

Other author is one of the designers of the XDS Modula-2 and Oberon-2 programming system. The XDS Modula-2 and Oberon-2 programming system allows, at first, to use the object-oriented resources of the second language, and secondly to transfer to the new operational environment earlier programmed non-object-oriented part of the package SIDM-2. Doing it is possible completely to adhere to the standard ISO, that makes possible the creation of the really portable simulation package. This new package ObSim-2 includes some modules on the language Modula-2, providing generation of the pseudo-random values and processes, matrix calculations and data processing. The Oberon-2 modules are intended for the description of frame and behavior of simulated systems. The compatibility of XDS multi-language programming system with main C++ compilers allows using a number of useful libraries also.

2 Advantages and Shortcomings of Modula-2 as Programming Language for DESS

In [2] Modula-2 was discussed as a basic language for the DESS design. It is well known [9] that any simulation language ought to provide the following features:

- means for the data organizing that provide simple and effective simulation;
- convenient means for the formulation and running the dynamic properties of a simulated system;
- possibility to simulate stochastic systems, i.e. procedures for generating and analysis of random variables and time series.

Now we can add that the object orientation is also of prime importance. Really, it could be said that OOP originated from simulation (refer to Simula-67! [11]).

It is clear to see that Modula-2 satisfies all demands but one: it has no standard modules for statistical support of a simulation process (no pseudo-random generators and data processing), but this is not a serious shortcoming as it is not very hard to design a special module with appropriate procedures. As for object-oriented properties, Modula-2 is an intermediate language. Modular concepts of this language allow to interpret some object-oriented features. Some extensions (TopSpeed for example []) include real class specifications. Moreover, Modula-2 has such valuable feature as quasi-parallel programming, that makes possible (with restrictions) process-oriented simulation. That explains why it was popular for DESS design in the late 1980s and early 1990s. Simulation package SIDM-2 also was programmed on Modula-2 because of its good convenience for the purpose.

This package provides the description of systems in the terms of discrete interactive processes (as in Simula) and events (similar to Simscript). This experience proves that Modula-2 is good for the purpose, but the TopSpeed extensions, first of all the object-oriented extension (classes) of the language were essentially used for rises of the efficiency of programs. The last circumstance has made the package hardly portable. At the same time SIDM-2 clearly shows that object-oriented features are of the prime importance for the efficiency and usability of simulation tools.

Processes are the part of Modula-2 that makes it good for the design of process-oriented DESS (Simula-like), but the process concept in Modula-2 has one severe shortcoming: it is simple to create any number of processes dynamically, but it is impossible to remove one from the program. According to the language description *end of any process is the and of a whole program.*

Strict typing is one of the main advantages of Modula-2. It allows a lot of possible mistakes to be avoided on the early stages of the program model development. At the same time *general pointer (type ADDRESS)* allows to create indirect transition of parameters to event procedures and processes. Using that is dangerous but effective. Thus, in SIDM-2 event procedures have the following type:

```
TYPE EVENT_PROC = PROCEDURE(ParField : ADDRESS);
```

When one ties event with procedure he use the special procedure `Event^.SetProc` while to designate the parameters one ought to create an example of structure designed for this special kind of event and then ties it with event procedure using another special procedure `Event^.SetPars`. Let us to illustrate this by the following example.

```
TYPE EVENT_PARS_POINTER = POINTER TO EVENT_PARS;
     EVENT_PARS = RECORD
                  Num_of_Device : CARDINAL;
                  Cust : POINTER TO Customer;
                  END; (* EVENT_PARS *)
     PoinEvent = POINTER TO EVENT;
VAR ArrPars : EVENT_PARS_POINTER;
    Arrival : PoinEvent;
 ...........
CLASS Event(Link);
 Pars : EVENT_PARS_POINTER;
 Proc : EVENT_PROC;
 ...........
 PROCEDURE SetPars(Pars : ADDRESS);
 PROCEDURE SetProc(Proc : EVENT_PROC);
 ...........
END Event;
 ...........
PROCEDURE New_Arrival(Pars : EVENT_PARS_POINTER); BEGIN
    UpdateStat(Device[Pars^.Num_of_Device]);
 ...........
END New_Arrival;
 ...........
NEW(Arrival); Arrival^.SetProc(New_Arrival);
 ...........
NEW(ArrPars);
ArrPars^.Num_of_Device:=k;
ArrPars^.Cust:=CurrentCust; Arrival^.SetPars(ArrPars);
Arrival^.Schedule(Time()+Negexp(1.0),TRUE);
 ...........
```

In this example the fragment of event procedure `New_Arrival` is presented that needs some parameters for execution. Special object `Arrival` of the class

`Event` is used for scheduling the event. It is clear to see that the procedural type `EVENT_PROC` allows to transfer parameters quit naturally. Unfortunately, as it was stated above, it is dangerous as it is not protected from any mistake in the type matching.

3 Modula-Like Simulation Systems

Some DESS have their own Modula-like programming languages. Most famous from them is MODSIM III [12]. The very fact of usage Modula-2 as a frame for the design of simulation languages proves its good features for the purpose. It is interesting that object-oriented means in MODSIM III are similar to those in the TopSpeed object-oriented extension of Modula-2 but are more powerful (for example it is possible to use multiple inheritance).

As in Modula-2 in MODSIM III *definition* and *implementation* modules are used. From [12] we can take the following example of the library module called `TextLib`:

```
DEFINITION MODULE TextLib;
  PROCEDURE Reverse(INOUT str : STRING);
END MODULE.

IMPLEMENTATION MODULE TextLib;
  PROCEDURE Reverse(INOUT str : STRING);
  VAR          { REVERSES THE INPUT STRING }
    k       : INTEGER;
    tempStr : STRING;
  BEGIN
    FOR k := STRLEN(str) DOWNTO 1
      tempStr := tempStr + SUBSTR(k, k, str);
    END FOR;
    str:=tempStr;
  END PROCEDURE; { Reverse }
END MODULE.
```

For the dynamic objects MODSIM III has operator `NEW` for creation and `DISPOSE` for destroying. That allows having an arbitrary number of dynamic objects during simulation run.

Of course, there are developed means for the event control and statistical support of a simulation process.

4 Can Oberon-2 Substitute Modula-2 in Simulation?

As it is well known, main differences between Oberon-2 and Modula-2 lay in object-oriented means [13,14].

We do not know about if Nicolas Wirth was acquainted with MODSIM III when he designed Oberon-2, but it is true that most of object-oriented means that were realized in MODSIM III are also presented in Oberon-2. Among them are:

1. multiple inheritance;
2. overriding methods;
3. concurrency.

Of most importance for the Simula-like simulation is the possibility to stop process (co-routine) without ending the whole program.

It is true, however, that some new (in comparison with Modula-2) features of Oberon-2 have hardened the programming of simulation models. Among these features is removing of the ADDRESS type from the language that makes impossible to use the approach to the parameters transition described above.

Simulation package ObSim-2 is the successor of SIDM-2. This package, as well as SIDM-2 [8], first of all is intended for simulation of systems, representable as a collection of inter-reacting discrete processes (like in Simula-67). At the same time the event-driven simulation means are included in this package.

It is possible to say, that the systems are considered that are representable as a collection of objects exchanging handle and information. The object is characterized by:

– data structure;
– rule of operations;
– operating schedule.

The data structure of the object includes its own data and the auxiliary information. This auxiliary information includes:

Number – individual number (usually is used for debugging);
Terminated – tag of a completeness of the process;
Name – a name of the object (important for the tracing mode);
TimeMark – the moment of creation;
EvNote – the reference to the event notice that is bounded with the object;
Proc – the reference to a co-routine implementing the operation rule of the object.

The operation rule of the object is represented by the quasi-parallel process that is realized or with the help of the Oberon-2 process (co-routine) or as the sequence of procedures (methods) of the object calls.

Under the operating schedule of the object an algorithm of choice the sequences of active phases of its operation in time is understood. The control transferring between objects is admitted only via a means of events planning.

According to mentioned above the following resources are included in the package:

– objects description;
– events planning;
– interaction of objects and storage of their data;
– the base means of statistical support of simulation experiment.

The special type of an event notice is used to provide the alternative (active phase of a process or procedure) mode of an event processing:

```
TYPE EventNotice*=RECORD(SS.Link);
   EvTime- : LONGREAL;      (* planned event time *)
   Host- : PoinEntObj;      (* if process *)
   END ; --EventNotice
```

Here the `SS.Link` is the class of links intended for placement into the event control list. The type `PoinEntObj` is of the special interest here:

```
TYPE
     PoinEntObj* = POINTER TO EntOrObject;
     PoinObj*    = POINTER TO Object;
     PoinEv*     = POINTER TO Event;

TYPE EntOrObject*=RECORD(SS.Link);
   _Name-      : ARRAY 16 OF CHAR;  (* for debugging *)
   No-         : LONGINT;           (* number, for debugging *)
   EvNotice-   : PoinEvNote;
   END; --EntOrObject

TYPE Object*=RECORD(EntOrObject);
 ............
TYPE Event*=RECORD(EntOrObject);
 ............
```

The event control procedure, based on the current type of `PoinEntObj` makes decision about to transfer the control to a process bounded with an object or call an event processing procedure.

5 Can Oberon-2 Win Competitive Struggle with C++?

It is a completed fact that C++ is now the main program development tool in a whole and in DESS design in particular. There are some reasons for this:

1. good object-oriented tools;
2. powerful compilers;
3. modern environments;
4. good acknowledged standard that allows to transfer programs between different platforms;
5. availability of a lot of different libraries for numerical analysis and computer graphics.

There is also one more subjective reason: somehow programming on C++ became "good fashion" among young programmers, may be because C is the main programming language in UNIX and to work under UNIX means to work in network environment that is prestigious also. Moreover, it is possible to say that there is some snobbery in membership of "C-programmers club".

By no we means do not deny good properties and efficiency of C and C++ in system development. At the same time we are aware of some their shortcomings. As hardest of them we can mention freedom of type conversion and weak protection from data access violation. C++ programs are not as easily readable as it is wanted also.

Oberon-2 as successor of Pascal can replace it in education (Pascal is still one of widely used programming languages in education) but it has some features suitable for the large program system design also. Really the only reason why it is not widely used is absence of the brand compiler on world market. Available compilers are mostly developed by universities and so have no good support and additional libraries.

6 XDS Modula-2 and Oberon-2 Programming System

The XDS Modula-2 and Oberon-2 programming system is a multi-language programming environment that allows to use native and second-part C programs also. This system includes ANSI Modula-2 and Oberon-2 compilers and a number of libraries that allows to control co-routine programming that is of a prime importance in simulation.

The possibility of using second-parties C-libraries in the XDS system allows to utilize the widely spread systems for the interface creation, that is very important for programming of modern DESS systems.

The first version of the package ObSim-2 was already used for simulation of digital networks with circuit switching that has shown greater convenience to the description of the simulated system also, than earlier used package SIDM-2. It is due to the object-oriented features of Oberon-2 mainly.

References

1. P. Waknis and J. Sztipanovits, DESMOD — a Discrete Event Simulation Kernel, *Mathl Comput. Modelling,* Vol. 14, pp. 93–96, 1990.
2. A.S. Rodionov, Using Modula-2 as Basic Programming Language for Simulation Systems, *System modeling-17, Bul. of the Novosibirsk Computing Center,* pp. 111–116, 1991. (in Russian)
3. A.S. Rodionov, Program Kernel for Simulation System Design on Modula-2, *Bulletin of the Novosibirsk Computing Center, Ser.: System modeling,* n.2 (20), pp. 75–82, 1994. (in Russian)
4. K. Ahrens and J. Fischer, DEMOS-M2 — a General Purpose Simulation Tool in Modula-2, *Syst. Anal. Model. Simul.,* Vol. 5, n. 3, pp. 215–221, 1988.
5. P. L'Ecuyer and N. Giroux, A Process-oriented Simulation Package Based on MODULA-2, *Proc. 1987 Winter Simulation Conference,* pp. 165–174, Atlanta, 1987.

6. J.A. Miller and O.R. Weyrich (Jr.), Query Driven Simulation Usung SIMODULA, *Proc. of the 22nd Annual Simulation Symp.,* pp. 167–181, Tampa, 1989.
7. R. Sharma and L.L. Rose, Modular Design for Simulation, *Software — Practice and Experience,* Vol. 18, n. 10, pp. 945–966, 1988.
8. A.S. Rodionov, Discrete Event Simulation Package SIDM-2, *Proc. 6th Int. Workshop "Distributed data processing" (DDP-98),* pp. 269–272, Novosibirsk, 1998.
9. P.J. Kiviat, Simulation languages, *In: T.H. Naylor ed., Computer Simulation Experiments with models of economic systems,* pp. 397–489, John Wiley & Sons, 1971.
10. *TopSpeed Modula-2 for IBM Personal Computers and Compatibles. Language and Library reference.* Clarion Software Corporation, 1992.
11. G. Birtwistle, SIMULA Begin, Petrocelli/Charter, N.-Y., 1973.
12. *MODSIM III. Object-oriented simulation. Tutorial,* CACI products company, 1996.
13. N. Wirth. From Modula to Oberon. The Programming Language Oberon (Revision Edition), *Departement Informatik, Eidgenössische Technische,* Zürich, 1990.
14. H. Mössenböck, *The Object-Oriented Programming in Oberon-2.* Springer, 1993.

Author Index